Mälardalen University Dissertations
No. 4

Towards A Static Cache Analysis for Whole Program Analysis

Xavier Vera

2002

MÄLARDALEN UNIVERSITY

Department of Computer Engineering
Mälardalen University

ISBN number: 91-88834-32-8
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

A DISSERTATION
PRESENTED TO THE FACULTY
OF MÄLARDALENS HÖGSKOLA
IN CANDIDACY FOR THE DEGREE
OF LICENTIATE IN ENGINEERING

With every waking breath I breathe
I see what life has dealt to me
With every sadness I deny
I feel a chance inside me die

Give me a taste of something new
To touch, to hold, to pull me through
Send me a guiding light that shines
Across this darkened life of mine

Breathe some soul in me
Breathe your gift of love to me
Breathe your life to lay 'fore me
Breathe to make me breathe

For every man who built a home
A paper promise for his own
He fights against an open flow
Of lies and failures we all know

To those who have and who have not
How can you live with what you've got
Give me a touch of something sure
I could be happy ever more

Breathe some soul in me
Breathe your gift of love to me
Breathe your life to lay 'fore me
Breathe to make me breathe
Breathe your honesty to me
Breathe your innocence to me
Breathe your word and set me free
Breathe to make me breathe

This life prepares the strangest things
The dreams we dream of, what life brings
The highest highs can turn around
To sow love's seeds on stony ground

Midge Ure – "breathe"

'cause time is what you make of it

To Helen, Anna and Marta
without whom this would not have been possible.

Abstract

Data caches have become very popular to overcome the gap between main memories and processors performance. Caches work well for programs with sufficient locality. Unfortunately, there are many programs that do not take advantage of them, thereby suffering large number of misses. Small modifications in the source code may change memory patterns, thereby altering the cache behaviour. If the code is modified properly, we can expect a high cache performance improvement. A detailed information about the number of misses and their causes is necessary to devise effective transformations. However, this information is very hard to obtain. This thesis describes a new method for describing the cache behaviour of whole programs.

Based on a new characterisation of data reuse across multiple loop nests, we present a method, a prototyping implementation and some experimental results for analysing the cache behaviour of whole programs with regular computations. Validation against cache simulation using real codes shows the efficiency and accuracy of our method. Our method can be used to guide compiler locality optimisations and improve cache simulation performance.

Acknowledgments

I sincerely thank my main supervisor Björn Lisper for giving me the chance to be a member of his research group. I also appreciate his support and his patience with my lousy English. Besides Björn, I would also like to especially thank Jan Gustafsson, my secondary advisor, for his useful comments on this document.

Special credit is due to Jingling Xue for his great hospitality during my visit. I learnt so much from our meetings and discussions.

I would like to express my gratitude to my roommates, Jan Carlson and Nerina Bermudo, for cheering me up in all those days when I am in bad mood, especially grumbling about the Swedish weather. I am grateful to Baran and Rikard, whose friendship was very important to me in those early days in Västerås.

Finally, I thank my parents and family. My gratitude to them is beyond what words can express.

Västerås, February 2002
Xavier Vera

Agraïments

Finalment! Quan un arriba aquí vol dir que tot el difícil ja està fet... Creu-t'ho això. Posat amb el LaTeX, babel i els accents en català, i anyoraràs el Word i començaràs a preguntar-te a qui li pot interessar uns agraïments en català.

Però em feia gràcia, ves, per allò de ser diferent. Així que després de fer més recerca per escriure aquestes quatre línies que per tota la tesi, un va i... comença a traduir els que tenia en anglès!!! Que lleig!!!

Au, torna a pensar mentires i afalagaments. Us pot semblar molt fàcil, però us asseguro que no ho és. A veure, per qui començo? El Toni? Per què l'hauria de posar? Perquè s'ha llegit la tesi? Perquè és collonut i ha estat l'únic que ha vingut a veure on rodaven "Doctor en Alaska"? Pels n-mil mails donant-me ànims quan vaig marxar, perquè és sempre genial tornar-lo a veure? No, si sembla que sí...

El Manel si que val. Ell sempre tan útil :-). Us puc assegurar que sense ell no crec que hagués arribat aquí (entre altres, perquè no tindria ni idea de LaTeX). O el Xavi, sempre preocupat en la distància (serà que les sueques li fan por), donant-me ànims i ajudant-me en els moments difícils que sempre tenim.

Ara que torno a rellegir el que he escrit (es que el LaTeX és d'un

intuïtiu!), m'adono que tot són pol** (qui deia que a Suècia no hi havia censura!?). Ben pensat, potser haver fet infermeria si només volia posar Maries i Cristines, no? Gràcies, Marta, per donar-me una altra oportunitat i ajudar-me a reestablir la comunicació H(o)uston-Marte. I a l'altra Marta (sí, aquí un d'idees fixes), per sempre trobar un moment entre panyals i potitos per escriure un correu. I la Julia, sempre tan comunicativa. O la Clara, per trobar-me i no perdre'm la pista després de milers de quilòmetres amunt i avall.

No voldria oblidar-me de tots aquells companys que vaig deixar a la UPC. A l'Antonio, per tot el que em va ensenyar i per la seva manera especial de motivar-me a millorar cada dia. Una abraçada a qui és un amic de debò: Jaume.

Ara només queda esperar que el del tribunal no parli català. Perquè ja sabem que deia l'eslògan: "Catalans, som 5.999.999... i el del tribunal!!".

I a tu. Per tot. Per res. Pels teus somriures. Per tot el que em dones i el que em prens, per fer que les il.lusions no siguin un miratge. Pel Temps. Perquè un dia la pluja ens acompanyarà.

JamagradariaBarcelona, Febrer 2002
Xavier Vera

Oz Acknowledgments

I couldn't believe it. I had the chance of visiting Australia! That country which is in the top of the world, where people only drink VB and play rugby without helmet. This was around September 2000 and I was out of words. First I'd like to thank Jingling Xue for inviting me. We'd never met before, but he believed we could get some work done. Second, my advisor Björn Lisper, for letting me go and letting me follow my hunches.

Those days in Sydney were something very special. I skipped the Swedish winter to dive into an oz summer and walk on those amazing beaches of Maroubra and Coogee. Did I forget to mention research?

I want to express my gratitude to all people from Warrane College. They made life in Sydney remarkable and a very pleasant experience. I can't write down all names, but I wish to thank R.V and Dave Curran (there's no way to understand an Irishman early in the morning... actually, there's no way to understand an Irishman) for their help during my first days. I'm especially glad of having met Lui, Jim, Joe, Steve, Marek and Richard (nice trip to Manly!). Some friendships go beyond the distance: Eugene, Max and Tino. *Y no, no me olvido de ti, Nick Santucci. Espero verte algún dia en Barcelona.*

Many thanks to all the administrative staff at UNSW, and in particular to the ss guys for helping me with the usual (too many) problems I have with computers.

Go Wallabies !
Västerås, February 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We introduce our work in the context of high-performance computer architecture. We present the reason why we decided to address this thesis, and the main goals we plan to achieve. Besides, we show how our current approach may fit in a real-time context. Finally, we state our contributions and give a road map of the rest of the document.

Use of recent architectural features such as pipelines, branch predictors and caches may result in significant performance improvements. However, their unpredictable nature results in an unpredictable behaviour. For instance, it is very hard to predict the execution time of a program with the presence of a cache memory, and it may even happen that a program runs faster without the cache rather than with it. Information such as the number of misses and where they occur would allow computing the execution time. If we went one step further and knew the causes behind those misses, we could optimise the code in such a way the program could take plenty advantage of the cache.

Data caches are widely used to bridge the increasing gap between processors and main memories speed. The effectiveness of a cache memory depends not only on the hardware structure, but also on the code generated by the compiler. Both programmers and compiler transformations often restructure the code to improve the performance of the memory system. While a large number of locality enhancement transformations exist [79], the models used for evaluating their benefits are often heuristic or approximate. For example, tiling [11, 37, 43, 61, 81, 82] and padding [39, 58] can reduce cache misses if appropriate tile sizes and pad sizes are chosen. However, even in these simple cases, no model has emerged as a widely acceptable solution. It is well-known that the optimal tile and pad sizes are sensitive to the problem size, array base addresses and cache parameters. We need better models that can determine not only the number of cache misses but also help us understand the causes behind these misses. These models can then be employed to guide various optimisations to reduce cache misses in a systematic manner.

Several approaches for analysing cache behaviour can be identified. Analysis has been performed traditionally either at compile-time using

heuristics or at run-time by means of simulators. Cache simulation techniques are very accurate and report a rich source of information. Based usually on trace-driven approaches [66], they are both time- and space-consuming and do not provide insights about the causes of the misses. Hardware counters [4], although fast and accurate, are architecture-dependent and do not provide too many insights about the causes of cache misses.

In the last few years, several compile-time analytical methods have been proposed to statically predict the cache behaviour of a program [13, 22, 28, 29, 69]. At this early stage, all these research efforts have focused on loop-oriented programs operating on arrays. Such a method consists of (a) a procedure for setting up mathematical formulas to characterise the cache misses in a program and (b) an algorithm for finding cache misses (and their causes, if required) from these formulas. These formulas describe the relationships among loop indices, array sizes, base addresses and the cache parameters for cache misses in the program. The better understanding of the causes of the misses helps reduce them in a systematical way, reasoning about the causes of the interferences. Some compiler-optimisations such as padding and tiling have been described by these methods [29, 68].

1.1 Cache Behaviour and Real Time Systems

In order to exploit these new architectural improvements in hard real-time systems, it is very important to know tight and safe upper bounds of the worst-case time (WCET) of tasks to be executed.

Calculation of a tight WCET bound of a program involves difficulties that come from the very characteristics of data caching. Even though some progress has been done when studying processors with instruction caches [5, 34, 47], few steps have done towards analysing data caches.

Alt, Ferdinand *et al* [3, 19] provide an estimation of WCET by means of abstract interpretation. As well as the usual drawbacks from abstract analysis (i.e., code-explosion and lack of accuracy), they only analyse memory references which are scalar variables. When providing experimental results, they only deal with instruction caches. White *et al* [73] propose a method for direct-mapped caches based on static simulation. They categorise static memory accesses into (i) first miss, (ii) first hit, (iii) always miss and (iv) always hit. Array accesses whose addresses can be computed at compile-time are analysed, but they fail to describe conflicts which are always classified as misses. Lim *et al* [48] present a method that tries to compute the WCET taking in account data caching. However, they only analyse static memory references (i.e., scalars), failing to study real codes with dynamic references (i.e., arrays and pointers). Kim *et al* [42] propose a method that extends and improves the previous method extending the analysis that classifies references as either static or dynamic. However, they deal neither with arrays nor with pointers (i.e., only detecting temporal locality). Besides, it is limited to basic blocks, without taking in account possible reuse among different subroutines or loop nests.

While this work aims to model cache behaviour in such a way we can devise cache compiler optimisations, we plan in the future to extend our analysis to fill the current gap in cache analysis for real-time systems. Our approach yields probabilistic cache miss ratios, which we expect to use for soft real-time software with large regular data (such as multi-

media applications and MPEG decoders). Furthermore, we introduce a feasible way of dealing with caches in WCET analyses where they are not currently considered.

1.2 Contributions

The Cache Miss Equations (CMEs) [28], by Ghosh *et al.*, represent an analytical method for analysing the cache behaviour of loop-oriented programs. These programs typically spend a considerable amount of time operating on arrays in loop nests. The CMEs describe the relationships among loop indices, array sizes, base addresses and the cache parameters for cache misses in a loop nest using a set of Diophantine equations. This characterisation makes it possible not only to obtain the number of misses but to understand the causes behind cache misses, and helps reduce these misses in a systematic manner. Besides, it allows analysing isolated iteration points without simulating all the previous. However, obtaining the *exact* number of cache misses from the CMEs is computationally expensive.

Our work generalises and extends Ghosh's CMEs to make whole program analysis possible.

The thesis is divided in the following sections. First, we summarise a statistical method to speed up the process of solving the equations, for both direct-mapped and set associative caches [69]. Second, we present an analytical method for analysing the cache behaviour of perfectly nested loops[1] containing IF statements with compile-time-analysable conditionals. These conditionals can contain ABS, MOD, MIN and

[1]Perfectly nested while-loops with bounded iterations for each while can be transformed into loop nests, thus they are also analysable.

MAX operators. In particular, we describe (for the first time) how to find required reuse vectors in the presence of IF statements and some complications that may arise when group reuse vectors are derived. We demonstrate how our method can also be used to analyse those imperfect loop nests that are sinkable by loop sinking. Third, we extend our analysis for predicting the cache behaviour of complete programs (i.e., multiple loop nests, call statements and subroutines) with regular computations [71]. Finally, we give some insights about how our analytical method can be used to implement different cache compiler optimisations, such as padding [68] and blocking [1].

The overall contributions are:

Reuse Analysis and Representation. By generalising traditional concepts such as iteration vectors and uniformly generated references for perfect loop nests, we introduce reuse vectors for quantifying reuse between references contained in multiple nests with the presence of IF statements. Our reuse representation includes Wolf and Lam's reuse vectors [77] as a special case, allowing potentially existing reuse-driven optimisations to be applied to multiple nests.

Whole-Program Analysis. We can handle programs with regular computations consisting of subroutines, call statements, IF statements and arbitrarily nested loops. In order to predict a program's cache behaviour *statically*, these programs must be free of data-dependent constructs such as variable loop bounds, data-dependent IF conditionals, indirection arrays and recursive calls.

Prototyping Implementation. Our prototyping system consists of components on normalising loop nests, inlining calls (abstractly), generating reuse vectors, sampling memory accesses and forming

and solving the equations for cache misses. The inlining component has not been implemented. In our experiments, the calls in all programs (if any) are inlined by hand.

Validation and Experimental Results. We have validated our method against cache simulation using programs from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. The largest program we have analysed, Applu from SPECfp95, has 3868 lines of FORTRAN code, 16 subroutines and 2565 references. Assuming a 32KB (direct-mapped, 2-way and 4-way, resp.) cache with a 32B line size, our method obtains the miss ratios with absolute errors (0.78%, 0.82% and 0.84%, resp.) in about 128 seconds while the cache simulation runs for nearly 5 hours on a 933MHz Pentium III PC. In comparison with the three recent compile-time analytical methods reported in [13, 22, 28], our method is the only one capable of analysing this scale of programs efficiently with accuracy.

1.3 Organisation

Chapter 2 presents the underlying model on which this work is based. We explain the different transformations we apply to the codes in such a way we get a new version suitable for our analysis. Some important concepts that help us to keep trace of the order in which the memory accesses are done are described. Finally, we introduce some basic statistical notions that are used to model the number of misses.

Once all the memory references are identified, we introduce our new approach to analyse whole program cache behaviour in the following chapters. Chapter 3 explains how we deal with call statements and subroutines. Chapter 4 describes the new characterisation of the reuse vec-

tors, whereas Chapter 5 shows how we formulate and solve the equations that yield detailed information about cache behaviour.

We present the results of our validation in Chapter 6, with extensive evaluations of different kernels and three whole programs. We have evaluated both the accuracy and feasibility of our analysis. Furthermore, we show in Chapter 7 how our model can be used to develop automatic cache compiler optimisations.

Finally, Chapter 8 discusses some related work. Chapter 9 contains a summary of the important results of this works. It also points out some future work that may be addressed using this thesis as a starting point.

At the end of the document, we have included some appendices that try to give some more insights for those readers that are not familiar with the high-performance architecture world. Appendix A is a complete background to cache memories from the point of view of high-performance. Cache memories are introduced as a solution to bridge the gap between main memories and processors performance, always from the point of view of achieving better and better performance. Some cache architectures are presented. Next, some terminology for program analysis used in this work is presented. Finally, we give the basic definitions that are necessary to understand locality analysis and the reuse vectors.

Appendix B contains some codes from the benchmarks we have used to evaluate our approach.

Chapter 2

Underlying Model

This chapter introduces how we represent the programs in order to analyse them. First, we show how we transform them, keeping the semantics, in such a way that we can extract all memory references and the order in which they are executed. Based on the iteration vectors, we describe how we compute the iteration space and how we attach subsets of it to the references. Finally, we model the cache behaviour using a statistical technique.

2.1 Cache-Architecture Model

We assume a uniprocessor with a k-way set associative data cache using LRU replacement (see Appendix A). In the case of *write* misses, we assume a fetch-on-write policy so that *writes* and *reads* are modelled identically. Our current analysis assumes a cache indexed using virtual addresses. Some systems index caches with physical addresses, making cache behaviour strongly dependent on page placement.

In a k-way set associative cache, a cache set contains k distinct cache lines. C_s and L_s denote the cache size and line size (in array elements), respectively. A *memory line* refers to a cache-line-sized block in the memory while a *cache line* refers to the actual block in which a memory line is mapped.

2.2 Program Model

Presently, we restrict ourselves to analysing FORTRAN77 programs with regular computations. FORTRAN has been the programming language used to write numerical applications for quite a few years. It does not allow recursion, and all the parameters in call statements are passed by reference. Unlike C, it stores arrays in memory in column-major order and it does not have pointers.

We can handle programs made up of subroutines consisting of possibly IF statements, call statements and arbitrarily nested loops. In order to predict at compile time a program's cache behaviour, the following restrictions are imposed:

- All loop bounds and array subscript expressions must be affine in terms of the enclosing loops.

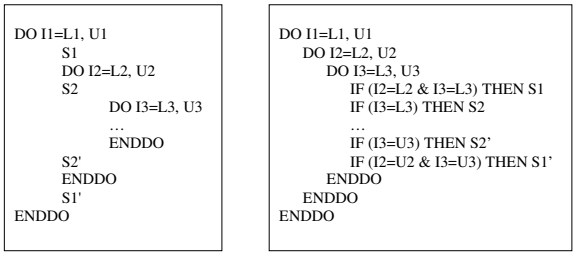
- The base addresses of all non-register variables including actual parameters (scalars or arrays) must be known at compile time.

- The sizes of an array in all but the last dimension must be known statically.

Our analytical method can deal with any IF conditionals involving loop indices and compile-time constants. In loop-oriented programs with regular computations, almost all data-independent conditionals are affine expressions of loop indices and compile-time constants involving possibly operators such as ABS, MOD, MAX and MIN. In all programs that we have analysed from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack, we have not found any single IF conditional that is data-independent but not also affine. Our program model excludes all and only data-dependent constructs (e.g., variable bounds, data-dependent IF conditionals and indirection arrays).

2.3 Analysis Model

Our model works for codes that have all the memory references in innermost loop nests. Besides, it needs all the innermost loops to have the same depth. In order to transform user codes in such a way they are suitable for being analysed, we first apply loop sinking, which moves all references to innermost loop nests. Thereafter, we apply loop normalisation, which adds dummy loop nests in order to have all the innermost loop nests in the same depth.



(a) Before sinking (b) After sinking

Figure 2.1: Sink transformation.

2.3.1 Loop Sinking

We present a strategy to analyse a subset of imperfectly nested loops. CMEs could only deal with perfectly nested loops. Thus, there are many important imperfectly nested loops that can not be analysed (as matrix multiplication with initialisation). Our strategy focuses on transforming imperfectly nested loops into perfectly nested loops with conditionals [76]. The loop nests we consider are those without two loops at the same level. Figure 2.1.a shows the form of those loop nests.

The technique consists in sinking all the statements to the innermost loop, obtaining a perfectly nested loop. Then, the statements are enclosed within an *IF* statement following Abu-Sufah's non-basic-to-basic-loop transformation [2]. The code obtained is shown in Figure 2.1.b. In the case code is already protected by an *IF* statement, this is another constraint that is taken in account. A loop transformation is called *legal* when the transformed code produces the same output as the original one. In order to obtain a loop nest semantically equivalent to the original one, the following conditions must hold [80]:

- The order of the references must be preserved.
- The innermost loop nest must be executed at least once, so if an iteration of a statement would have executed, then it is executed in the transformed program.

This transformation may change the sequence in which memory references are executed. Moreover, it may introduce some new memory references that are necessary to evaluate the new conditionals. In order to obtain the cache behaviour of the original program, we only analyse the original memory references and assume loop indices are register allocated. Thus, we apply this transformation for analysis purpose, but we do study the original code.

2.3.2 Loop Nest Normalisation

Loop normalisation consists in adding different DO statements that iterate only one iteration, in such a way that all innermost loop nests have the same depth. After normalisation, all loop nests are n-dimensional, and, in addition, all loop variables at depth k are normalised to I_k.

Unlike loop sinking, there is no restriction for applying this transformation, since the new statement cannot modify the semantics of the program.

2.3.3 Iteration Vectors

After applying loop normalisation, we have statements distributed among innermost loops. Unlike the case where all statements are in the same nest (the execution order of the statements is given by the order we find them in the code), now we need a special mechanism to describe their execution order.

A particular instance of a statement S (known as an *iteration* or *iteration point*) of the enclosing loop nest is identified by a $2n$-dimensional *iteration vector* of the form $\vec{\imath} = (\ell_1, I_1, \ell_2, I_2, \ldots, \ell_n, I_n)$, where

- $\vec{L} = (\ell_1, \ell_2, \cdots, \ell_n)$ is the *loop label* (*vector*) for the innermost loop containing S, and
- $\vec{I} = (I_1, I_2, \ldots, I_n)$ is the *index vector* consisting of the indices of the n loops enclosing S.

Figure 2.2 lists the iteration vector for each statement in the example. It is not difficult to see how the iteration vectors are derived in general.

Let $\vec{\ell}_L$ be the loop label for loop nest L. Since we have applied loop normalisation, all the loop labels are n-dimensional, where n is the depth of the deepest loop nest. The i-th entry of the loop label, if defined, is the order of the loop nest in the i-depth[1]. Using this formulation, we obtain the following loop labels for the example in Figure 2.2:

L_1:	$\vec{\ell}_{L_1}$=(1,*)
L_2:	$\vec{\ell}_{L_2}$=(1,1)
L_3:	$\vec{\ell}_{L_3}$=(1,2)
L_4:	$\vec{\ell}_{L_4}$=(2,*)
L_5:	$\vec{\ell}_{L_5}$=(2,1)

As usual, the set of all iterations for a particular loop nest is called the *iteration space* of that nest (see Appendix A, Section A.2.1).

In a sequential execution, all iteration points are executed in lexicographical order. The usual lexicographic order operators $\prec$, $\preceq$, $\succ$ and $\succeq$ are used later.

[1] This can be seen as the numbering of sections and subsections in a paper.

L_1: DO $I_1 = \ldots$	Iteration Vector
L_2: DO $I_2 = \cdots$	
S_1: $B(I_2 - 1, I_1) = \cdots$	$(1, I_1, 1, I_2)$
L_3: DO $I_2 = \ldots$	
S_2: $\cdots = B(I_2, I_1)$	$(1, I_1, 2, I_2)$
L_4: DO $I_1 = \cdots$	
L_5: DO $I_2 = \cdots$	
S_3:$B(I_1, I_2) = \cdots$	$(2, I_1, 1, I_2)$

Figure 2.2: The iteration vectors for statements.

2.3.4 Reference Iteration Spaces

The *reference iteration space (RIS)* of a reference R, denoted RIS_R, is defined as the set of iteration points where the reference is accessed. If a reference is not guarded by a conditional, its RIS is the entire iteration space of the enclosing loop nest. Otherwise, the RIS can be a subspace of that iteration space.

While we can analyse complex IF conditionals (resulting in non-convex RISs), the RISs in practical programs are found to be simple. In all programs analysed from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack, we have not found a single case that is not affine.

If a reference is guarded by affine conditionals (containing possibly .AND., .OR. or .NOT. operators), the corresponding RIS can always be expressed as a finite union of convex polytopes in $\mathcal{Z}^n$. Such a RIS can be manipulated by the Omega library [56] and its volume computed using methods [14, 33, 56] for various purposes. Figure 2.3 depicts three commonly occurring cases. An example is given in Figure 2.4. The three highlighted z references will be used later for illustrations. Ref_1 is not

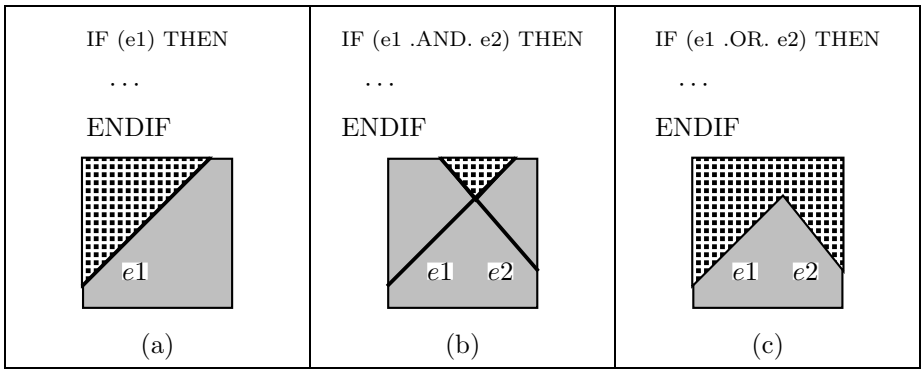
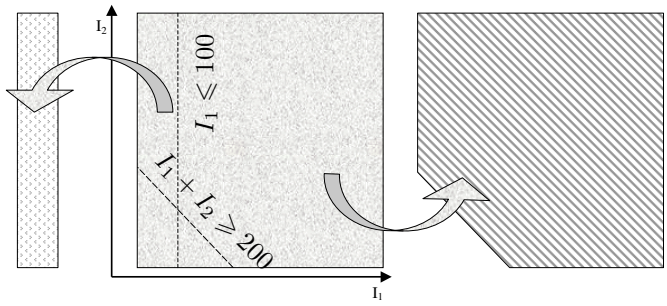


Figure 2.3: Some commonly occurring RISs (in dotted areas).

```
PROGRAM COND
PARAMETER (N = 512, M = 512)
REAL*8 a(N+1,M+1), b(N+1,M+1), z(N+1,M+1)
REAL*8 vnew(N+1,M+1), unew(N+1,M+1)
DO I1 = 1,N
  DO I2 = 1,M
    a(I1+1,I2) = b(I1+1,I2)+[z(I1+1,I2+1)] ≙ Ref1
    IF (I1+I2.GE.200) THEN
      vnew(I1,I2+1) = 1+[z(I1,I2+1)] ≙ Ref2
    ENDIF
    IF (I1.LE.100) THEN
      unew(I1,I2) = b(I1,I2)+[z(I1,I2)] ≙ Ref3
    ENDIF
  ENDDO
ENDDO
END
```

Figure 2.4: A running example.



(a) Ref_3 (b) Ref_1 (c) Ref_2

Figure 2.5: RISs of the three z references in Figure 2.4.

guarded while Ref_2 and Ref_3 are guarded by conditionals that are affine expressions of loop indices.

Figure 2.5 displays the RISs for the three z references highlighted in Figure 2.4, which are all convex.

Our analytical method can deal with any IF conditionals involving loop indices and compile-time constants. These are the conditionals that can be analysed at compile-time without relying on any runtime information about the conditionals involved. However, data-dependent conditional expressions such as *a(i,j).EQ.0* are beyond our current method and their analysis is part of our future work (see Section 9.1).

2.4 Statistical Model

We use a statistical approach to determine the cache behaviour. This allows us to obtain accurate results in a feasible time.

First we will present the basic statistical concepts used to model the cache behaviour, and secondly the model itself.

2.4.1 Discrete random variables

Let $\mathcal{S} = (\Omega, \mathcal{A}, P)$ be a probability space (where Ω is the sample space, $\mathcal{A} \subset \wp(\Omega)$[2], and P is the probability function). We may define random variables (RV) [16] $X : \Omega \to \mathbb{R}$ over $\mathcal{S}$. We review two of most studied discrete RV which have been used in our model.

Let X be a real discrete random variable:

- We say that it follows a Bernoulli distribution ($X \sim B(p)$) when the image set has only two elements. Bernoulli RVs describe the random experience in which only two things can happen: success or miss. We define $\mathcal{T} \subset \Omega$ as the set of results obtained that we consider as 'success'. Thus:

$$\begin{array}{rcl} X : \Omega & \longrightarrow & \mathbb{R} \\ \omega & \longmapsto & \begin{cases} 0 \iff w \notin \mathcal{T} \\ 1 \iff w \in \mathcal{T} \end{cases} \end{array}$$

  The probability $P[X = 0]$ is p. Therefore, the probability $P[X = 1]$ is $q = 1 - p$, since $p + q$ must be 1.

- Binomial distribution (represented by $X \sim Bin(n, p)$) models phenomena where n different and independent experiments modelled by Bernoulli-RV take place. This RV represents the number of successes.

  Once $\mathcal{T} \subset \Omega$ is defined, we obtain:

$$\begin{array}{rcl} X : \Omega^n & \longrightarrow & \mathbb{R} \\ (\omega_1, \ldots, \omega_n) & \longmapsto & card\{i | \omega_i \in \mathcal{T}\} \end{array}$$

[2] $\wp(X)$, the power set of X, is the set of all the possible subsets of X

The probability $P[X = k]$, $k = 0 \ldots n$ represents the probability that k experiments over the n succeed. Thus,

$$P[X = k] = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{(n-k)}.$$

2.4.2 Modelling the Cache Behaviour with Random Variables

We are interested in finding the number of misses that a program results in (said $\#misses$). In order to compute it, we model the problem in the following way: for each reference we define a RV that returns the number of misses.

We may model the behaviour of a reference using a Binomial-RV, where the different experiments consist in taking an iteration point and checking whether it results in a miss.

Next, we prove that this RV actually follows a Binomial distribution. For each memory instruction, we may define a Bernoulli-RV $X \sim B(p)$ as follows:

$$\begin{array}{rccl} X : Iteration & Space & \longrightarrow & \mathbb{R} \\ & \vec{\imath} & \longmapsto & \{0, 1\} \end{array}$$

such that $X(\vec{\imath}) = 1$ if the memory instruction results in a miss for iteration $\vec{\imath}$, $X(\vec{\imath}) = 0$ otherwise. Note that X describes the experiment of choosing an iteration point and checking whether the memory instruction produces a miss for it, and p is the probability of success. The value of p is $p = \frac{\#m}{N}$, where N is the number of times this instruction is executed and $\#m$ the number of misses.

Then, we repeat the experiment N times, using different iteration points in each experiment, obtaining $X_1, \ldots, X_N$ different RV-variables. We note that:

- All the $X_i, i = 1 \ldots N$ have the same value of p.
- All the $X_i, i = 1 \ldots N$ are independent[3].

The variable $Y = \sum X_i$ represents the total number of misses in all N experiments. This new variable follows a binomial distribution with parameters Bin(N,p) [16] and it is defined over all the reference iteration space.

2.4.3 Estimation of Parameters

Although a random variable describes a certain property, it may sometimes happen that it is impossible to obtain the parameters that define the RV. This may happen in the cases where population is very large, as in our case, where RISs may have millions of iteration points. In order to overcome this limitation, a subset of the population we try to describe can be analysed and the results obtained can be inferred to the population. Now, we explain how the parameters that describe a Binomial-RV can be inferred.

Let $X \sim Bin(n, p)$, and assume that p (the probability of success) is unknown. We obtain an approximation of p evaluating the behaviour of a subset of the population (called sample). The RV that describes the property we are interested in is then computed for the sample. Finally, we infer the sample-RV parameters to the population-RV.

[3] We assume that the iteration space is sampled in an independent way.

Let $\mathcal{Q} \subset \Omega^n$ be the sample, $N = card(\Omega^n)$ and $k = card(\mathcal{Q})$. The value $\hat{p}$ is defined as

$$\hat{p} = \frac{successes \in \mathcal{Q}}{k}$$

If the sample is randomly chosen among the population, the RV that describes the behaviour of the sample is $Y \sim Bin(k, \hat{p})$, and we have that[4]:

$$\frac{(\hat{p} - p)}{\sqrt{\frac{pq}{k}}} \sim N(0, 1)$$

provided that the sample does not contain repeated elements and the following conditions hold [16]:

- $\frac{k}{N} \leq 0.05$
- $\hat{p}k \geq 5$ and $1 - \hat{p}k \geq 5$
- $k \geq 30$

Once a confidence level[5] is chosen, a confidence interval for the value of p is given by the following expression[6]:

$$p \in \hat{p} \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}(1 - \hat{p})}{k}}$$

[4] Z∼N(0,1) is the Normal or Gauss distribution

[5] e.g: if the percentage is 95%, it represents that for 95 out of every 100 different samples, $\hat{p}$ will belong to the confidence interval

[6] α = 1-*confidence*

Chapter 3

Call Statements

In the previous chapter we have shown how we obtain all the necessary information to analyse codes with multiple loop nest. Since we want to analyse whole programs, we should deal with call statements and subroutines. This chapter introduces our Abstract Inlining Technique, which applied to all different calls, allows us to analyse whole programs.

3.1 Gentle Introduction to FORTRAN Subroutines

FORTRAN supports two kind of procedures, which must not be recursive. Subroutines begin with *subroutine name(arg1, arg2,...)*. Thereafter, the types of all arguments and local variables should be declared. Subroutines are called from another routine with the command *call name(arg1, arg2,...)*.

Functions begin with *function name(arg1, arg2,...)*, then the types of *name*, all arguments and local variables should be declared.

All arguments are passed using "call by reference" (like VAR arg in Pascal or &arg in C/C++). Changing the value of an argument in the subroutine changes the value of the corresponding variable in the calling program. Actual arguments in the calling program may have different type or dimensioning from in the declaration in the function or subroutine. No type checking is done during compiling or run time.

Arguments in a function or subroutine can be declared as arrays with variable size. This means that the address of the first array element is passed to the subroutine, and calls to subsequent array elements access subsequent memory addresses. For two dimensional arrays the first array index must be known. Local variables cannot be arrays with variable size (since FORTRAN does not have dynamic memory allocation).

3.2 Abstract Inlining

In an attempt to analyse exactly a program containing call statements, we perform an *abstract inlining* for a call whenever possible. We do not actually generate the inlined code. We only need to obtain the in-

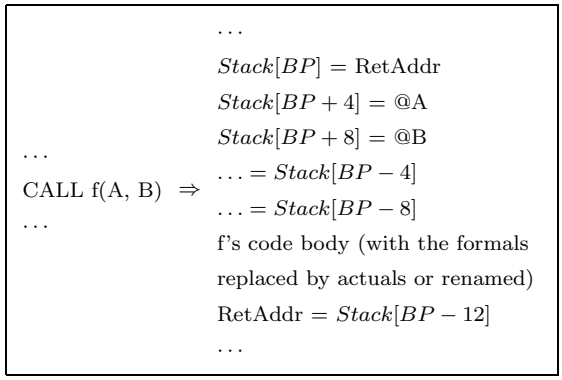


Figure 3.1: Abstract inlining of a subroutine call.

formation required for analysing the inlined code. Each subroutine is associated with an *abstract function* consisting of the information about the memory accesses to the run-time stack, its code body (i.e., its loop nests with references), and local variable and formal parameter declarations. As shown in Figure 3.1, every call to a subroutine is abstractly inlined by replacing the call with the information in the abstract function associated with the subroutine.

The calling conventions used for a program are compiler- and architecture-dependent. Figure 3.1 depicts one such a convention for 32 bit machines. *Stack* denotes the run-time stack modelled as a one-dimensional array of an infinite size. If SP is 0 initially, its value is known at compile time at every call site due to the absence of recursive calls. The base address of *Stack*, if unknown at compile time, has to be obtained at run time. Then *Stack* is treated just like an ordinary array reference. For large programs, the impact of these stack accesses is insignificant.

Not every call can be inlined according to Table 3.1. To analyse a call

Program	Actual Parameters			Calls	
	P-able	R-able	N-able	Total	A-able
Tomcatv	0	0	0	0	0
swim	0	0	0	5	5
su2cor	503	87	0	150	150
hydro2d	122	0	19	82	82
mgrid	68	0	35	23	2
applu	79	0	0	23	23
apsi	1601	0	210	186	118
fppp	83	0	3	17	16
turb3D	759	0	75	111	86
wave5	591	2	110	171	127
CSS	2489	0	8	965	965
LWSI	140	0	19	28	18
MTSI	186	0	2	63	63
NASI	236	0	237	75	41
OCSI	620	0	48	244	209
SDSI	189	18	49	129	103
SMSI	321	0	41	53	38
SRSI	242	0	176	50	13
TFSI	137	0	91	44	13
WSSI	836	127	7	185	179
TOTAL	9202	234	1130	2604	2251
%	87.09	2.21	10.89	100	86.44

Table 3.1: Statistics for the actual parameters and calls in SPECfp95 and Perfect benchmarks.

exactly, our method needs to know at compile time the base addresses of all its actual parameters. Let AP be an actual parameter that is either a scalar or an array variable or a subscripted variable with an affine data access expression and FP be its matching formal parameter.

AP is *propagateable* if, after inlining, every reference to FP can be replaced by a reference to AP. This allows the reuse to AP both in the caller and in all the callees to be potentially exploited. In Column "P-able", we consider AP as propagateable if FP is a scalar, or one-dimensional array or if both AP and FP are arrays of the same dimensionality with matching sizes in all but the last dimension.

Example. Let {A(N,M), A(I1,I2)} be the actual parameter (A is an array whose dimensions are N and M), and {S(N,M), S(I3,I4+1)} the formal parameter. We are interested in the memory address accessed by the formal parameter. Let SOF be the size of the elements of the array. The memory address is given by the expression

$$@S + (I3 - 1) * SOF + (I4 + 1 - 1) * SOF * N$$

Since parameters are passed by reference, $@S$ is the memory address of the actual parameter. This is, $@A + (I1 - 1) * SOF + (I2 - 1) * SOF * N$. Substituting, we obtain

$$@A + (I1 - 1) * SOF + (I2 - 1) * SOF * N + (I3 - 1) * SOF + (I4 + 1 - 1) * SOF * N$$

Finally, grouping, we obtain that it is analysable:

$$@A + (I1 + I3 - 2) * SOF + (I2 + I4 - 1) * SOF * N = A(I1 + I3 - 1, I2 + I4)$$

End Example.

AP is *renameable* if, after inlining, every reference to FP can be replaced by a reference to AP' such that AP and AP' have the same

base address (i.e., $@AP = @AP'$). The propagateable actuals are not also classified as renameable. In Column "R-able", we consider AP as renameable if the sizes of all but the last dimension for AP and FP are known statically. This still allows the reuse between the references to FP in the same subroutine to be exploited.

Example. Let {A(N,M), A(I1,I2)} be the actual parameter (A is an array whose dimensions are N and M), and {S(N',M',P), S(I3,I4,2)} the formal parameter. We are interested in the memory address accessed by the formal parameter. Let SOF be the size of the elements of the array. The memory address is given by the expression

$$@S + (I3 - 1) * SOF + (I4 - 1) * SOF * N' + (2 - 1) * SOF * N' * M'$$

Since parameters are passed by reference, $@S$ is the memory address of the actual parameter. This is, $@A + (I1 - 1) * SOF + (I2 - 1) * SOF * N$. Substituting, we obtain

$$@A + (I1 - 1) * SOF + (I2 - 1) * SOF * N + (I3 - 1) * SOF + (I4 - 1) * SOF * N' +$$

$$+ (2 - 1) * SOF * N' * M'$$

Finally, grouping and renaming S by S1, we obtain:

$$@A + (I1 - 1 + (I2 - 1) * N + I3 - 1) * SOF + (I4 - 1) * SOF * N' +$$

$$+ (2 - 1) * SOF * N' * M' =$$

$$= S1(I1 - 1 + (I2 - 1) * N + I3, I4, 2)$$

End Example.

In Column "N-able", the actuals that are neither propagateable nor renameable, known as *non-analysable*, are represented. The propagateable and renameable actuals are potentially *analysable* since all references to FP can be analysable if affine.

A call can be abstractly inlined, i.e., is potentially *analysable*, if all its actuals are analysable. Table 3.1 shows that we can inline 86.44% of calls from SPECfp95 and Perfect benchmarks. These statistics are obtained by examining only a call and its callee.

Figure 3.2 serves to illustrate the inlining of a code segment (which may have out of array bound accesses if loop bounds are not chosen properly). The inlined code does not compile (dimensions of the arrays declared in the main program should be statically known) but can be analysed by our method. Hence, the name abstract inlining.

Finally, system calls (to I/O subroutines and intrinsic functions) are not inlined. The memory accesses inside are not accounted for. These calls can be inlined if their abstract functions are known.

```
REAL*8 X, A, B
DIMENSION A(10,10), B(20,20)
DO I1 = ...
  DO I2 = ...
    A(I1,I2) = ···
    CALL f(X, A, B, B(I1,I2))
    CALL g(A(I1,I2), A(1,I2), B)

SUBROUTINE f(Y, C, D, S)
REAL*8 Y, C, D, S
DIMENSION C(10,10), D(400), S(10,10,*)
DO I3 = ...
  DO I4 = ...
    C(I3,I4-1) = Y + D(I3-1+20*(I4-1))
    S(I3,I4,2) = ···

SUBROUTINE g(E, F, T)
...
REAL*8 E, F, T
DIMENSION E(10,10), F(10), T(100,4)
DO I3 = ...
  DO I4 = ...
    E(I3,I4) = F(I4) - T(I3,I4)
```

```
REAL*8 X, A, B, B1, B2
DIMENSION A(10,10), B(20,20)
C THE FOLLOWING LINE DOES NOT COMPILE
DIMENSION B1(10,10,*), B2(100,4)
DO I1 = ...
  DO I2 = ...
    A(I1,I2) = ···
    DO I3 = ...
      Do I4 = ...
        A(I3,I4-1) = X + B(I3-1+20*(I4-1))
        B1(I1+10*(I2-1)+I3-1, I4, 2) = ···
    DO I3 = ...
      Do I4 = ...
        A(I1+I3-1, I2+I4-1) = A(I4,I2) - B2(I3,I4)
```

Figure 3.2: Propagation and renaming of actual parameters. All actuals but the last are propagated. The last actuals in both calls are renamed to $B1$ and $B2$, respectively. After inlining, $@B = @B1 = @B2$.

Chapter 4

Reuse Vectors

Once we have all the references identified and the different RISs built, it is time to analyse the locality of the program. In this chapter we explain how we devise the new reuse vectors, describing the locality among different RISs.

Caches are effective only when programs exhibit sufficient data locality in their memory accesses. Therefore, accurate approaches that estimate the locality are very useful. Different approaches provide trade-offs between: accuracy, speed, flexibility (i.e., adaptability to different memory configurations) and information provided.

In this chapter, the concept of reuse vectors is introduced (for a more intuitive introduction, see Appendix A) and some previous work on computing reuse vectors recalled. Then we describe how we compute reuse vectors. We generalise Wolf and Lam's reuse framework [77, 82] to calculate reuse vectors across different RISs, including multiple nests and conditionals. We also add additional spatial reuse vectors to capture the reuse spanning two adjacent columns of an array. Finally, some discussions on our approach are provided.

$Mem\_Line_R(\vec{i})$ ($Cache\_Set_R(\vec{i})$) (see Section A.1.2 in Appendix A) denotes the memory line (cache set) to which the memory address accessed by reference R at iteration $\vec{i}$ is mapped.

Let $Mem\_Addr_R(\vec{i})$ be the memory address of the reference R at iteration $\vec{i}$. We have:

$$\begin{aligned} Mem\_Line_R(\vec{i}) &= \lfloor Mem\_Addr_R(\vec{i})/L_s \rfloor \\ Cache\_Set_R(\vec{i}) &= Mem\_Line_R(\vec{i}) \mod \mathcal{N} \end{aligned}$$

where L_s is the cache line size (in bytes) and $\mathcal{N} = C_s/k$ is the number of cache sets.

4.1 Uniformly Generated References

We recall the definition of uniformly generated reference [77]. Let n be the depth of a loop nest, and d be the dimensions of an array R. Two

references $R(f(\vec{\imath}))$ and $R(g(\vec{\imath}))$, where f and g are indexing functions $\mathbb{Z}^n \rightarrow \mathbb{Z}^d$, are called uniformly generated if

$$f(\vec{\imath}) = H\vec{\imath} + \vec{c}_f \quad g(\vec{\imath}) = H\vec{\imath} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

After loop nest normalisation (see Section 2.3.2), $\vec{I} = (I_1, I_2, \ldots, I_n)$ is the index vector of all n-dimensional loop nests. The concept of uniformly generated references for perfect loop nests [24, 77] can be carried over to multiple nests. All z references in our running example (see Figure 2.4) are uniformly generated, whereas there are two uniformly generated reference sets in Figure 2.2: $\{B(I_2 - 1, I_1), B(I_2, I_1)\}$ and $\{B(I_1, I_2)\}$.

4.2 Recalling Reuse Vectors

The concept of reuse vectors was introduced by Wolf and Lam in [77] as a mathematical representation to determine the direction and distance of data reuse between uniformly generated references. Let R_p (p for 'producer') and R_c (c for 'consumer') be two uniformly generated references $A(H\vec{\imath} + \vec{c}_p)$ and $A(H\vec{\imath} + \vec{c}_c)$, respectively. Let $\vec{r} \succeq \vec{0}$ be an integer vector. R_c at iteration $\vec{\imath}$ (with the memory access $A(H\vec{\imath} + \vec{c}_c)$) reuses potentially from R_p at $\vec{\imath} - \vec{r}$ (with the memory access $A(H(\vec{\imath} - \vec{r}) + \vec{c}_c)$) if

$$Mem\_Line_{R_c}(\vec{\imath}) \quad = \quad Mem\_Line_{R_p}(\vec{\imath} - \vec{r})$$

Then $\vec{r}$ is said to be a *reuse vector*. It represents a *potential* reuse in the cache between the two memory accesses since the memory line touched in the cache at the first access (at $\vec{\imath} - \vec{r}$) may have been evicted from the cache before it gets reused at the second access (at $\vec{\imath}$). As is customary,

$\vec{r}$ is *temporal* (reusing the same element) if the following equality also holds:

$$Mem\_Addr_{R_c}(\vec{\imath}) \quad = \quad Mem\_Addr_{R_p}(\vec{\imath} - \vec{r})$$

and *spatial* (reusing the same cache line but not the same element) otherwise. In addition, the reuse is said to be a *self-reuse* if R_c and R_p are identical and a *group-reuse* otherwise. Thus, there are four kinds of reuse: self-temporal, group-temporal, self-spatial and group-spatial.

Wolf and Lam [77] discuss how to compute reuse vectors for perfect loop nests with straight-line assignments, assuming all RISs are the entire iteration space. By quantifying the reuse of a loop nest using a vector space spanned by (elementary) reuse vectors, they apply unimodular and tiling transformations to improve parallelism and locality in the nest. Later, Xue and Huang [82] describe an extension to allow non-elementary reuse vectors to be represented exactly. If the columns of every array are aligned at the memory line boundaries, Wolf and Lam's reuse framework provides all reuse vectors required. Otherwise, some extra reuse vectors are needed to represent cross-column reuse cases.

Consider our running example (see Figure 2.4), where z is a 2-D array of size $(N+1) \times (M+1)$. Suppose that a cache line has four array elements and that $z(N-1,1)$, $z(N,1)$, $z(N+1,1)$ and $z(1,2)$ resides in a common memory line in that order. For Ref_3, i.e., $z(I_1, I_2)$, the access $z(N-1,1)$ at iteration $(N-1,1)$ may potentially reuse this memory line in the cache touched by the access $z(1,2)$ at the earlier iteration $(1,2)$. This reuse is described by the self-spatial reuse vector $(N-1,1)-(1,2) = (N-2,-1)$, which is not captured by Wolf and Lam's framework. For details on computing Wolf and Lam's reuse vectors, see [77].

4.3 Group Reuse Among Different RISs

Let R_p and R_c be two uniformly generated references. Let R_p be the producer $A(M\vec{I} + \vec{m_p})$ nested inside the innermost loop labelled by $(\ell_1^p, \ell_2^p, \cdots, \ell_n^p)$ (see Section 2.3.3) and R_c be the consumer $A(M\vec{I} + \vec{m_c})$ nested inside the innermost loop labelled by $(\ell_1^c, \ell_2^c, \cdots, \ell_n^c)$, where $\vec{I} = (I_1, I_2, \ldots, I_n)$.

Consider temporal reuse between R_p and R_c. Iterations $\vec{\imath}_1$ of R_p and $\vec{\imath}_2$ of R_c reference the same data whenever $M\vec{\imath}_1 + \vec{m_p} = M\vec{\imath}_2 + \vec{m_c}$, that is, when $M(\vec{\imath}_1 - \vec{\imath}_2) = \vec{m_p} - \vec{m_c}$.

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be a solution to:

$$M\vec{x} \quad = \quad \vec{m_p} - \vec{m_c} \tag{4.1}$$

and

$$\vec{r_t} = (\ell_1^c - \ell_1^p, x_1, \ell_2^c - \ell_2^p, x_2, \ldots, \ell_n^c - \ell_n^p, x_n)$$

such that $\vec{r_t} \succeq 0$. Then $\vec{r_t}$ is a *temporal reuse vector* from R_p to R_c.

In FORTRAN, all arrays are column-major. Let $\vec{y} = (y_1, y_2, \ldots, y_n)$ be a solution to:

$$\begin{aligned} &M'\vec{y} = \vec{m'_p} - \vec{m'_c} \\ &|M_1\vec{y}| < L_s \end{aligned} \tag{4.2}$$

but not a solution to (4.1), where M_1 is the first row of M, and every primed term is obtained from its corresponding term in (4.1) with its first row or entry removed. Let

$$\vec{r_s} = (\ell_1^c - \ell_1^p, y_1, \ell_2^c - \ell_2^p, y_2, \ldots, \ell_n^c - \ell_n^p, y_n)$$

such that $\vec{r_s} \succeq 0$. Then $\vec{r_s}$ is a *spatial reuse vector* from R_p to R_c.

If a memory line spans two adjacent columns of an array, we will add spatial reuse vectors to capture such reuse. The spatial reuse vectors of

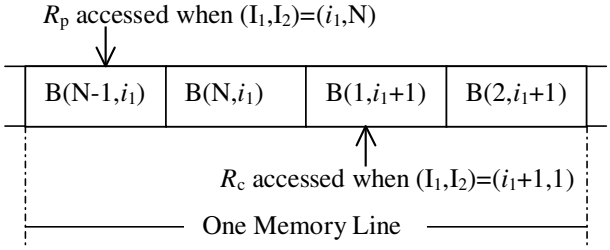


Figure 4.1: Spatial reuse across array columns ($L_s = 4$).

this second kind are added individually depending on the iteration space shapes and cache parameters used.

Let us derive reuse vectors for the first two references to B in Figure 2.2. Let R_p be $B(I_2 - 1, I_1)$ nested in the inner loop labelled by $\vec{L_p} = (1, 1)$ and R_c be $B(I_2, I_1)$ nested inside the inner loop labelled by $\vec{L_c} = (1, 2)$. The subscript expressions for both references are affine:

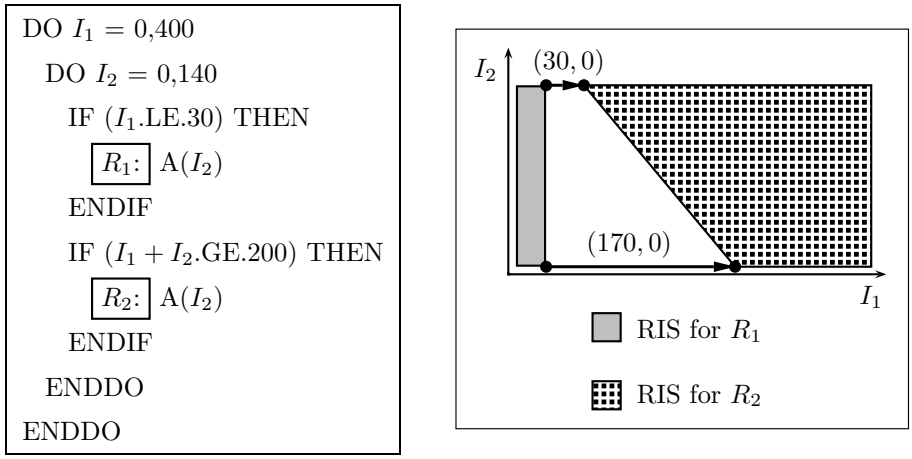
$$\begin{aligned} M\vec{I} + \vec{m_p} &= \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right] \\ M\vec{I} + \vec{m_c} &= \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} I_1 \\ I_2 \end{smallmatrix}\right] + \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right] \end{aligned}$$

In this case, the equation (4.1) becomes:

$$\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right]$$

which has the unique solution $(0, -1)$. Thus, the unique temporal reuse vector from $B(I_2 - 1, I_1)$ to $B(I_2, I_1)$ is $(0, 0, 1, -1)$. To find the spatial reuse vectors spanning a single column of B, we solve:

$$\begin{aligned} \left[\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] &= 0 \\ \left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] &< L_s \end{aligned}$$



(a) Code (b) RISs

Figure 4.2: Derivation of group-reuse vectors.

which is the instance of the equation (4.2) for this case. Thus, all these vectors have the form $(0, y_2)$. We would add the following spatial reuse vectors:

$$(0, 0, 1, -2), (0, 0, 1, -3), \ldots, (0, 0, 1, -(L_s - 1))$$

Finally, we would need reuse vectors $(0, 1, 0, 1 - N)$ to capture the reuse for the elements at the end of one array column and the beginning of the next column. This is illustrated in Figure 4.1.

4.4 Discussion

If a reference is guarded by an IF conditional, its RIS may not be the entire iteration space of the enclosing loop nest. This causes complications only in the derivation of group temporal reuse vectors. The self

temporal and spatial reuse vectors for a reference are defined and derived without a need to refer to its RIS. As for the group reuse vectors from R_p to R_c, a possible way to handle irregular RISs is to generate all potential ones conservatively. In the case of group temporal reuse, there can be infinitely many reuse vectors to from some facets of RIS_{R_p} to some facets of RIS_{R_c}.

Consider an extreme example illustrated in Figure 4.2. R_2 (the reusing reference) at every point (I_1, I_2) on the left boundary of its RIS may reuse R_1 (the reused reference) at the point $(30, I_2)$ on the right boundary of R_1's RIS along the symbolic group-reuse vector $(I_1 - 30, 0)$. If we ignore the two conditionals to analyse the reuse between the two references, the group-reuse vector $\vec{r} = (0, 0)$ will describe correctly the reuse from R_1 to R_2. When the miss equations for R_2 are formulated, the two conditionals must be taken into account. Then this reuse vector will be ignored since the two RISs do not overlap. As a result, the number of cache misses for R_2 on the left boundary of its RIS may be over-estimated. For practical applications, such an over-estimation is negligible because (a) the over-estimation occurs only on a facet of a RIS (e.g., the left boundary of R_2's RIS) and (b) the underlying reference may reuse on the facet via other reuse vectors. In the example, R_2 may reuse from itself along the self-spatial reuse vector $(1, -1)$. Thus, only a small fraction of these boundary points are mis-predicted.

In our implementation, these reuse vectors are ignored. Our extensive validation confirm that an overestimation of cache misses thus caused is negligible since (a) we overestimate only on some facets of RIS_{R_c} and (b) R_c may reuse on the facets by other reuse vectors (usually self reuse vectors).

Chapter 5

Cache Behaviour Analysis

Reuse vectors describe the locality and potential reuse among different references. But reuse does not necessarily result in a hit. This chapter shows how we describe cache behaviour based on the reuse description. First, we set up a collection of equations that characterise those iteration points that do not result in a miss. Secondly, we present two different methods that yield the miss ratio from those equations.

Following the same classification as the CMEs, we divide our miss equations into two groups : *compulsory or cold (miss) equations* and *replacement (miss) equations*. *Cold* misses represent the first time a memory line is touched while *replacement* misses are those accesses that result in misses because the cache lines that would have been reused were evicted from the cache before they get reused. Note that replacement equations represent both capacity and interference misses.

In this section, we present the miss equations as a specification of the cache misses in a loop nest. We then discuss two algorithms for finding cache misses from these equations. In particular, our replacement miss equations are formulated and solved differently from those in the CMEs [28] since the involved RISs can be different. We also describe an algorithm for computing efficiently the volume of a RIS for sampling purposes.

5.1 Forming Equations

Let $\vec{r}$ be a reuse vector from the producer reference R_p to the consumer reference R_c. We want to find out if R_c at iteration $\vec{i}$ can reuse the cache line accessed by R_p at $\vec{i}-\vec{r}$. Let R_i be an *intervening* reference such that the access of R_i at some iteration point $\vec{j}$ between $\vec{i}-\vec{r}$ and $\vec{i}$[1] may be mapped to the same cache set as the access of R_p at $\vec{i}-\vec{r}$. If that happens, a *set contention* occurs between the access of R_p at $\vec{i}-\vec{r}$ and the access of R_i at $\vec{j}$. In a k-way set associative cache, it takes k distinct set contentions to evict the cache line touched by the access of R_p at $\vec{i}-\vec{r}$.

We give below the miss equations that can be further analysed to

[1] In lexicographical order.

determine if the access of R_c at $\vec{\imath}$ is a miss or hit, assuming the single reuse vector $\vec{r}$ from R_p to R_c and the single intervening reference R_i.

5.1.1 Cold Equations

The cold equations for R_c along $\vec{r}$ represent the iteration points where the memory lines are brought to the cache for the first time:

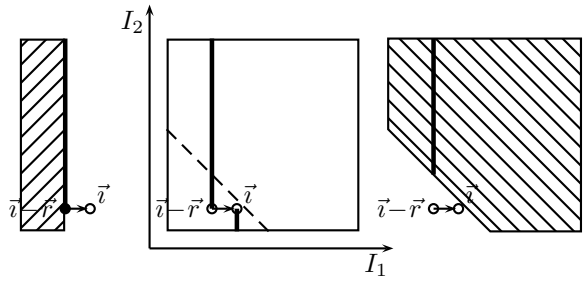
$$\begin{gathered}\vec{\imath} \in RIS_{R_c} \\ \text{and} \\ \big(\vec{\imath} - \vec{r} \notin RIS_{R_p} \\ \text{or} \\ Mem\_Line_{R_c}(\vec{\imath}) \neq Mem\_Line_{R_p}(\vec{\imath} - \vec{r})\big)\end{gathered}$$

If $\vec{r}$ is temporal, the inequality is false and thus redundant.

5.1.2 Replacement Equations

The *replacement equations* for R_c along $\vec{r}$ are to investigate if R_c at iteration $\vec{\imath}$ can reuse the cache line that R_p accessed at iteration $\vec{\imath} - \vec{r}$ subject to the set contentions caused by the memory accesses from R_i. This equations only describe cache set contentions, and we rely on the solver to check whether they result in a miss or not. R_i may cause a cache set contention at all intervening points executed between $\vec{\imath}$–$\vec{r}$ and $\vec{\imath}$:

$$\begin{gathered}Mem\_Line_{R_c}(\vec{\imath}) = Mem\_Line_{R_p}(\vec{\imath} - \vec{r}) \\ \vec{\imath} \in RIS_{R_c} \\ \vec{\imath} - \vec{r} \in RIS_{R_p} \\ Cache\_Set_{R_c}(\vec{\imath}) = Cache\_Set_{R_i}(\vec{\jmath}) \\ \vec{\jmath} \in J_{R_i}\end{gathered}$$



(a) $R_i = Ref_3$ (b) $R_i = Ref_1$ (c) $R_i = Ref_2$

Figure 5.1: The interference sets with the three z references when $R_c = R_p = Ref_1$ along $\vec{r} = (1, 0)$ for the running example. For illustration purposes, the point $\vec{\imath} \in RIS_{Ref_1}$ being analysed is chosen such that $\vec{\imath} \notin RIS_{Ref_2}$ and $\vec{\imath} \notin RIS_{Ref_3}$. In each case, the interference set consists of the solid line(s) and $\vec{\imath}$ or $\vec{\imath} - \vec{r}$ if the corresponding point is a fat point.

where J_{R_i} denotes the set of all these intervening iteration points, called the *interference set* for R_c along $\vec{r}$, and is specified precisely by:

$$J_{R_i} \quad = \quad \{\vec{\jmath} \in RIS_{R_i} \mid \vec{\jmath} \in \ll \vec{\imath} - \vec{r}, \vec{\imath} \gg\}$$

where '$\ll$' is '[' if R_i is lexically after R_p and '(' otherwise and '$\gg$' is ']' if R_i is lexically before R_c and ')' otherwise. A reference is neither lexically before nor lexically after itself. Figure 5.1 shows the interference sets with the three z references (see our running example in Figure 2.4) when Ref_1 is analysed along its self-spatial reuse vector $\vec{r} = (1, 0)$.

5.2 Finding Cache Misses

Miss equations contain precise information about cache behaviour, but obtaining the information such as number and causes of the misses is not straightforward. In the following sections, we recall some ways to interpret the miss equations and present two methods that give the number of misses of a reference.

5.2.1 Overview

Each equation represents a convex polyhedron[2] in $\mathbb{R}^n$ [9, 26], where n depends on the type of equation. The integer points inside each convex polyhedron represent the potential cache misses. We may consider two different ways of computing them, either by solving the equations or checking whether a point is solution or not.

Analytical method

In this section we will give an analytical description of the solution set of the original CMEs. This solution set represents the cache misses of a reference, and its volume the number of misses. The method is based on the following two theorems [26]:

- **Theorem 1:** The set of all misses of a reference along a reuse vector is given by the union of all the solution sets of the equations corresponding to that reuse vector.

Given a reference and a reuse vector, an iteration point produces a miss if it is either a compulsory or a replacement miss.

[2]The original work only deals with convex iteration spaces. In our approach this is not true anymore.

- **Theorem 2:** The set of all miss instances of a reference is given by the intersection of all the miss-instance sets along the reuse vectors.

Given a reference, an iteration point results in a hit if it exploits the locality of at least one of the reuse vectors.

Thus, given a reference R with m reuse vectors and n_k equations for the k^{th} reuse vector, the polyhedron that contains all the iteration points that result in a miss is [26]:

$$Set\_Misses = \cap_{k=1}^{m} \cup_{j=1}^{n_k} Solution\_Set\_Equation_j$$

The number of polyhedra that must be counted is 2^s [69], making this problem infeasible due to its huge computing time.

Traversing the iteration space

The second method is based on the fact that every iteration point can be studied independently from the rest of the iteration space.

Given a reference, all iteration points are tested independently, studying the equations in order: from the equations generated for the shortest reuse vector to the equations generated by the longest one [27].

Let us consider a reference R. We study the reuse vectors in a lexicographical ascendent order. After one reuse vector has been treated, some iteration points will be identified as resulting in a miss or a hit. Others rest undetermined. The iteration points are studied as follows:

- If an iteration point is a solution to a cold equation, the reuse along the reuse vector $\vec{r}$ is not realised in this iteration point, but we cannot take any definitive decision about the character of this

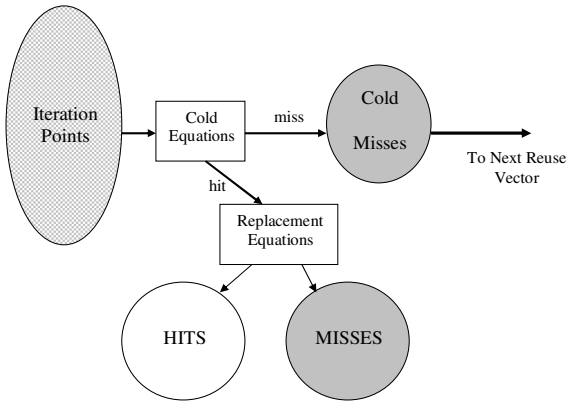


Figure 5.2: Studying iteration points through a reuse vector.

iteration point until all reuse vectors have been studied. Thus, this point will be considered as undetermined.

- If an iteration point is not a solution to any of the cold equations, it will be declared as a miss if it is a solution to a replacement equations, and as a hit if it is no solution to any equation either.

The algorithm stops when all iteration points have been characterised. Figure 5.2 shows the analysis for a particular reuse vector.

Counting the total number of solution is a very time-consuming process, since we may expect very large RISs. This makes it rather impractical to be included in product-compilers. However, we can use sampling techniques (see Section 2.4) to capture the distribution of misses and use the relative results for both guiding compiler-optimisations and obtaining the number of misses.

5.3 FindMisses and EstimateMisses

Figure 5.3 gives two algorithms for obtaining the cache misses from a whole program, consisting of multiple references and reuse vectors in multiple loop nests. Both *FindMisses* and *EstimateMisses* analyse each reference by going through its reuse vectors in lexicographical order $\prec$. If an iteration point is a solution to the cold equations along the current reuse vector $\vec{r}$, its behaviour is indeterminate and will be examined further using the other reuse vectors later in the list. Otherwise, the iteration point is classified either as a hit or a miss using the replacement equations along $\vec{r}$. After all reuse vectors have been tried, the remaining indeterminate iteration points are cold misses.

Our replacement equations represent only cache set contentions. In a k-way set associative cache, it takes k distinct cache set contentions to cause a cache line to be evicted from the cache set. There will be a cache miss only when k distinct solutions are found [28, 70].

FindMisses analyses all iteration points in a RIS and is practical only for programs of small sizes [28, 70]. *EstimateMisses* analyses a sample of a RIS and is capable of analysing programs significantly more efficiently with a controlled degree of accuracy. *EstimateMisses* expects the user to enter values to the two parameters: the *confidence percentage* c and the *confidence width* w, *where* $0\% < c \leqslant 100\%$ *and* $0 < w < 1$ (see Section 2.4). The two input values determine the size of the sample taken from RIS_R and also impose a lower bound on $|RIS_R|$. If a RIS is too small to achieve (c, w), we either use the default values $(c', w') = (90\%, 0.15)$ (which requires a sample size of 72 points and $|RIS_R| \geqslant 1440$ [69]) or analyse all points in RIS_R (when $|RIS_R| < 1440$). The meanings of c and w are such that if we run *EstimateMisses* many times, the

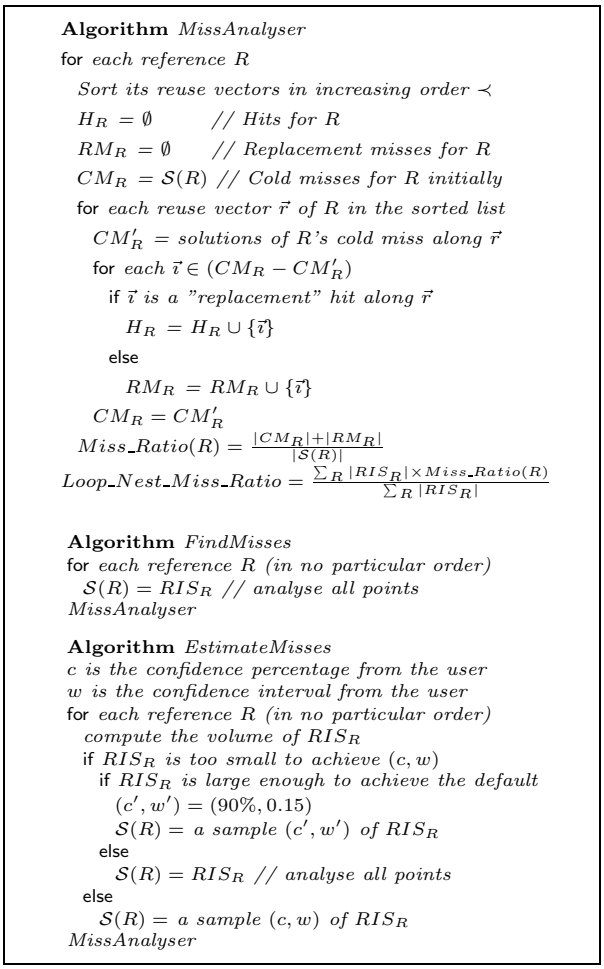


Figure 5.3: Two algorithms for computing the cache misses from the cold and replacement miss equations.

real miss ratio for each R obtained in c of these runs will lie in the interval $[Miss\_Ratio(R) - w/2, Miss\_Ratio(R) + w/2]$. However, this interpretation does not apply to the miss ratio for the entire loop nest given in line 15. In all our experiments, real and estimated miss ratios are close (see Chapter 6).

Thus, the statistical sampling technique used requires the size of every RIS to be calculated. Our algorithm for computing the volume of a RIS is described as follows. If the IF conditions guarding a reference form a union of convex polyhedra, then the corresponding RIS is a union of convex polyhedra because the iteration space is convex. The number of points contained in such a RIS is calculated by slicing the RIS recursively into regions of lower and lower dimensions until eventually every region is either empty or a (one-dimensional) union of line segments so that the points in the region can be counted easily. This algorithm, while exponential in terms of the dimension of the iteration space, is very efficient for practical programs with simple loop bounds and affine conditionals. Other methods for computing the volume of a convex polytope also exist [14, 56].

If a reference R is guarded by some non-affine conditionals, then RIS_R can be arbitrarily complex. There is not any general method for computing the volume of RIS_R. In our implementation, we compute the volume of such a RIS by proceeding as before with all non-affine conditionals ignored and then count only those points that satisfy all non-affine conditionals. This simple extension has not been used in our experiments since we have not found any data-independent conditionals that are not affine in any programs analysed.

Chapter 6

Validation

In this chapter we evaluate extensively our approach. We present results for different kernels, isolating IF conditionals and multiple loop nests in different tests. Finally, we put everything together and we show the accuracy and feasibility of our approach for analysing whole programs.

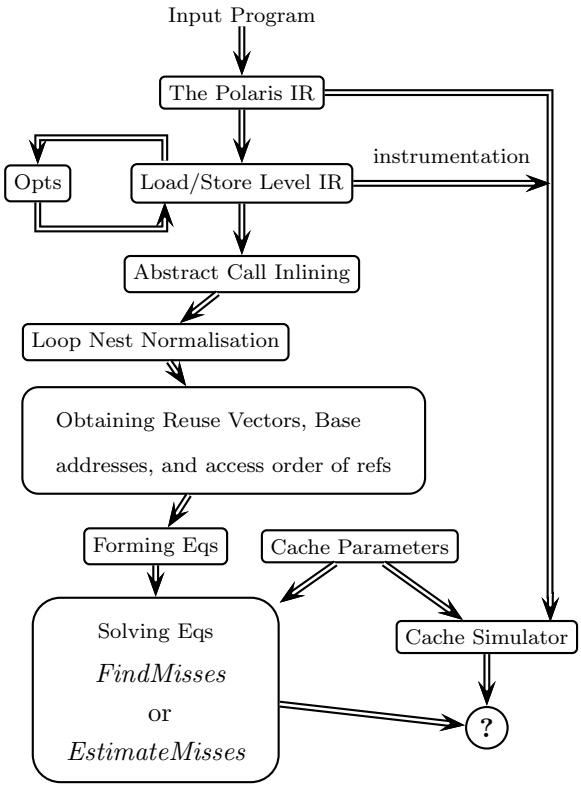


Figure 6.1: A framework for analysis and evaluation.

6.1 Prototyping Implementation

Figure 6.1 depicts the structure of our prototyping system for finding cache misses and validating the accuracy of our method against a cache simulator. We have implemented our method in the Coyote Miss Equations solver [8]. The component *Opts* optimises the program and allocates variables to registers or memory. The reuse vectors, the base

addresses of variables and the relative access order of memory references are obtained from a load-store low-level internal representation (IR), which is produced from the Polaris[1] IR [18] of the program using Ictineo[2] [6]. The inlining component is currently being implemented. The same information obtained is fed to both our algorithms and the cache simulator used.

6.2 Analysing IF Conditionals

We have analysed a range of programs from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack. We report our experimental results for four single loop nests.

Unless otherwise specified, we assume a 32KB cache with a 32B cache line size. The execution times are all obtained on a 933MHz PentiumIII PC. All simulation results are obtained using a trace-driven simulator.

6.2.1 Loop Nest Kernels

We present our experimental results for our running example (see Figure 2.4) and the three loop nests given in Appendix B. The problem sizes are those as specified in the programs. In all cases, we assume a cache of C_s=32KB with L_s=32B bytes per cache line and 8 bytes per array element. Therefore, every cache line has four array elements.

[1]The Polaris compiler takes a Fortran77 program as input. It can transform this program so that it runs efficiently on a parallel computer.

[2]Ictineo library is built on the top of Polaris. Its high-level internal representation allows doing both high and low-level transformations and optimisations in a unified way.

Prog.	Cache	#Cache Misses		Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*F.M.*	Sim.	*F.M.*	Err	Sim.	*F.M.*
COND	direct	1164004	1164004	81.69	81.69	0.00	0.53	55.62
	2-way	1157335	1157335	81.22	81.22	0.00	0.58	105.00
	4-way	1157335	1157335	81.22	81.22	0.00	0.58	180.64
LU	direct	81440	85193	6.13	6.41	0.28	0.32	67.09
	2-way	57441	70643	4.32	5.31	0.99	0.33	71.04
	4-way	61278	77461	4.61	5.83	1.22	0.34	77.83
MM	direct	287697	287700	7.17	7.17	0.00	1.02	55.24
	2-way	262699	262702	6.55	6.55	0.00	1.04	59.31
	4-way	262699	262702	6.55	6.55	0.00	1.10	65.01
LWSI	direct	802	816	0.16	0.16	0.00	0.17	1.50
	2-way	802	816	0.16	0.16	0.00	0.17	14.36
	4-way	802	816	0.16	0.16	0.00	0.17	30.69

Table 6.1: Cache misses for (C_s, L_s) =(32KB,32B) and execution times for *FindMisses* (*F.M.*).

FindMisses

This algorithm finds the cache misses from the miss equations by analysing all iteration points (i.e., all memory accesses) in the loop nest. It is computationally expensive for large iteration spaces since it performs essentially a compile-time cache simulation of the loop nest. However, this algorithm can be used ideally to evaluate the accuracy of our method, in particular, our reuse vector analysis. Table 6.1 compares *FindMisses* and a cache simulator for caches of different associativities. The absolute error between the miss ratios in both cases in all examples is negligible. The execution times in all cases indicate that analysing all points is too expensive to be used at compile-time in guiding compiler optimisations.

Some further discussions are provided below.

COND Both *FindMisses* and the simulator yield the same results in all cache configurations.

LU *FindMisses* over-estimates the cache misses in all cache configurations used. The mis-predictions are due to the lack of reuse vectors to describe the reuse that exists among the non-uniformly generated references: *a(j,i), a(i,i), a(j,k)* and *a(i,k)*. For example, *a(i,i)* accesses *a(1,1)* and *a(j,i)* accesses *a(2,1)* at the same iteration $(1, 1, 2)$. Both accesses are to the same cache line. The lack of a reuse vector to describe this particular reuse results in the memory access $a(1, 1)$ to be classified incorrectly as a miss. To validate this assumption, we ran *FindMisses* by adding four additional group-spatial reuse vectors: $(0, 0, 0)$ from $a(j, i)$ to $a(i, i)$, $(0, 1, 0)$ from $a(i, i)$ to $a(j, i)$, $(0, 0, 0)$ from $a(j, k)$ to $a(i, k)$ and $(0, 1, 0)$ from $a(i, k)$ to $a(j, k)$. The cache misses obtained for the "direct", "2-way" and "4-way" cases have been reduced to 81553, 64704 and 71200, respectively. As a result, the absolute errors in these cases have been reduced to 0.00, 0.55 and 0.75, respectively.

MM *FindMisses* over-estimates the number of misses in all three cases by a margin of three. The three mis-predictions are due to the lack of reuse vectors to describe the spatial reuse between references *b(i,k)* and *c(k,j)*. The base addresses for b and c are 230136 and 310136, respectively. Thus, the memory addresses of $b(98, 100)$, $b(99, 100)$, $b(100, 100)$ and $c(1, 1)$ are 310112, 310120, 310128 and 310136, respectively. This implies that all four elements reside in the same memory line (starting at 475). A simple analysis shows that the access $b(i, 100)$ at iteration $(i, 1, 100)$ reuses this memory line brought into the cache by the access $c(1, 1)$ at iteration $(i, 1, 1)$,

Prog.	Cache	Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*E.M.*	Err	Sim.	*E.M.*
COND	direct	81.69	81.29	0.40	0.53	0.40
	2-way	81.22	80.92	0.30	0.58	0.64
	4-way	81.22	80.92	0.30	0.58	0.97
LU	direct	6.13	6.49	0.36	0.32	0.68
	2-way	4.32	5.18	0.86	0.33	0.70
	4-way	4.61	5.73	1.12	0.34	0.69
MM	direct	7.17	7.18	0.01	1.02	0.12
	2-way	6.55	6.44	0.11	1.04	0.11
	4-way	6.55	6.44	0.11	1.10	0.13
LWSI	direct	0.16	0.15	0.01	0.17	0.35
	2-way	0.16	0.15	0.01	0.17	0.50
	4-way	0.16	0.15	0.01	0.17	0.65

Table 6.2: Miss ratios for (C_s, L_s) =(32KB,32B) and execution times of *EstimateMisses* (*E.M.*) ($c = 95\%$ and $w = 0.05$).

where $98 \leqslant i \leqslant 100$. Due to the lack of reuse vectors, these three accesses to b are classified as misses.

LWSI The transformed program by loop sinking consists of five conditionals some of which are quite complex. In our experiments, the five scalars ($zero, wsin, wcos, z$ and xs) are treated as one-dimensional arrays of single elements each, which happen to reside in four different memory lines with other array variables. *Find-Misses* over-estimates the cache misses by 14 in all three cases due to the lack of reuse vectors to describe the reuse among all these memory lines.

Program	Cache	Miss Ratio		Abs.	Exe.T (secs)	
		Sim.	*E.M*	Err	Sim.	*E.M.*
COND N=M=1000	$\mathcal{C}\#1$	82.42	82.22	0.20	2.19	0.60
	$\mathcal{C}\#2$	94.15	93.82	0.33	2.22	0.63
	$\mathcal{C}\#3$	31.47	31.10	0.37	2.16	0.61
LU N=1000	$\mathcal{C}\#1$	19.33	19.99	0.66	349.41	0.83
	$\mathcal{C}\#2$	44.71	44.77	0.06	387.5	2.24
	$\mathcal{C}\#3$	6.23	6.44	0.21	353.26	1.74
MM N=M=400	$\mathcal{C}\#1$	13.97	13.68	0.29	68.77	0.13
	$\mathcal{C}\#2$	50.03	50.03	0.00	74.82	0.72
	$\mathcal{C}\#3$	6.33	6.04	0.29	68.21	0.14
LWSI ns=50 natoms=1000	$\mathcal{C}\#1$	36.79	37.29	0.50	244.7	0.59
	$\mathcal{C}\#2$	78.35	78.97	0.62	262.4	1.26
	$\mathcal{C}\#3$	15.27	15.37	0.10	244.32	0.3

$\mathcal{C}\#1$: $(C_s, L_s, k) = (64\text{KB}, 16\text{B}, \text{direct})$
$\mathcal{C}\#2$: $(C_s, L_s, k) = (32\text{KB}, 8\text{B}, 2)$
$\mathcal{C}\#3$: $(C_s, L_s, k) = (128\text{KB}, 32\text{B}, 4)$

Table 6.3: Cache misses for three different cache configurations and execution times of *EstimateMisses (E.M.)* ($c = 95\%$ and $w = 0.05$).

EstimateMisses

This algorithm finds cache misses from the miss equations of a reference by taking a sample from its RIS. Table 6.2 shows the accuracy and efficiency of *EstimateMisses* using a 95% confidence percentage with an interval width of 0.05. In all but one case, the difference between the estimated miss ratio and the real one is less than 1.0. The difference in the exceptional 4-way LU case is 1.12. This is due to the lack of reuse vectors for describing the reuse among the non-uniformly generated references as discussed previously. To validate this assumption, we ran

EstimateMisses by adding the same four additional group-spatial reuse vectors as before: $(0,0,0)$ from $a(j,i)$ to $a(i,i)$, $(0,1,0)$ from $a(i,i)$ to $a(j,i)$, $(0,0,0)$ from $a(j,k)$ to $a(i,k)$ and $(0,1,0)$ from $a(i,k)$ to $a(j,k)$. The miss ratios for the loop nest obtained for the "direct", "2-way" and "4-way" cases have been reduced to 6.35, 4.85 and 5.42, respectively. As a result, the absolute errors in these cases have been reduced to 0.22, 0.53 and 0.81, respectively.

The execution times in all cases are less than a second on a 933MHz Pentium III PC.

Table 6.3 evaluates *EstimateMisses* further for different problem sizes on different cache configurations.

6.3 Whole Program Analysis

We present our results for three isolated kernels containing multiple loop nests, and three whole programs.

6.3.1 Multiple Loop Nest Kernels

We evaluate the accuracy of our method by comparing *FindMisses* (which analyses all iteration points) with a cache simulator. Table 6.4 presents the results in both cases for caches of different associativities. In all but one case, our method obtains exactly the same miss ratio as the simulator. In the exceptional case, we overestimate slightly the miss ratio by 0.05.

Figure B.2 shows the three kernels used:

- **Hydro** is a 2-D explicit hydrodynamics from Livermore (kernel 18). *FindMisses* and the simulator yield the same results in all

Program	Cache	#Cache Misses		%Miss Ratio		Abs. Error	Execution Time (s)
		Sim.	*F.M.*	Sim.	*F.M.*		
Hydro (KN=JN=100)	direct	52603	52603	14.12	14.12	0.00	1.07
	2-way	52603	52603	14.12	14.12	0.00	1.35
	4-way	42703	42703	11.47	11.47	0.00	1.64
MGRID (M=100)	direct	1518879	1518879	9.49	9.49	0.00	91.29
	2-way	1424038	1424038	8.90	8.90	0.00	99.45
	4-way	1424038	1424038	8.90	8.90	0.00	100.70
MMT (N=BJ=100 & BK=50)	direct	145671	147075	4.82	4.87	0.05	43.09
	2-way	171647	172592	5.68	5.71	0.03	47.06
	4-way	246980	247744	8.18	8.20	0.02	57.44

Table 6.4: Cache miss ratios for 32KB caches with a 32B line size from *FindMisses* and a cache simulator.

cases.

- **MGRID** is a 3-D loop nest from MGRID. Again *FindMisses* and the simulator agree on their results in all cache configurations.

- **MMT** is a 3-D blocked loop nest taken from [22] that computes the matrix multiplication A and B^T. The two references to WB are not uniformly generated due to the transposition of B. Being unable to exploit their reuse, *FindMisses* over-estimates the cache miss ratios in all three cases slightly. Due to transposition, the degree of reuse between the two references is rather minimal. The inaccuracy lies in the incompleteness of reuse information rather than our method itself.

Table 6.4 indicates that *FindMisses*, while being capable of finding

Program	Cache	Abs. Error	Execution Time (secs)
Hydro (KN=JN=100)	direct	0.05	0.27
	2-way	0.05	0.32
	4-way	0.05	0.36
MGRID (M=100)	direct	0.36	0.19
	2-way	0.32	0.22
	4-way	0.32	0.22
MMT (B=BJ=100 & BK=50)	direct	0.23	0.10
	2-way	0.37	0.10
	4-way	0.37	0.11

Table 6.5: Cache misses from *EstimateMisses* for 32KB caches with a 32B line size ($c = 95\%$ and $w = 0.05$).

exactly cache miss numbers, does so at the expense of large execution times.

Table 6.5 shows the accuracy and efficiency of *EstimateMisses* using a 95% confidence with an interval of 0.05 for all references in the program. In all cases, the absolute errors are less than 0.4 and the execution times less than 0.5 seconds. Note that *EstimateMisses* yields only the miss ratio for a program. The actual miss ratio of each kernel is available in Table 6.4.

6.3.2 Whole Programs

We evaluate *EstimateMisses* against a simulator using three programs from SPECfp95 detailed in Table 6.6. In each case, we have succeeded in abstractly inlining all the calls and obtained one loop nest for the program. In addition, all actual parameters are propagateable, meaning that the references to every actual can be potentially exploited across

	Tomcatv	Swim	Applu
#lines	190	429	3868
#subroutines	1	6	16
#call-statements	0	6	27
#references	79	52	2565

Table 6.6: Three whole programs.

Program	Cache	Miss Ratio		Abs.	Exe.T	Sim.T
		Sim.	*E.M*	Err	(secs)	(secs)
Tomcatv	direct	11.42	11.02	0.40	0.30	3676.2
	2-way	11.40	11.0	0.40	0.37	3750.3
	4-way	11.41	11.0	0.41	0.58	3860.2
Swim	direct	7.26	7.01	0.25	2.47	8136.0
	2-way	6.98	6.73	0.25	2.63	8281.1
	4-way	7.24	6.97	0.27	3.23	8425.8
Applu	direct	6.95	7.73	0.78	127.31	17089
	2-way	6.60	7.42	0.82	127.6	17155
	4-way	6.56	7.40	0.84	127.5	17278

Table 6.7: Cache misses from *EstimateMisses* for 32KB caches with a 32B line size ($c = 95\%$ and $w = 0.05$).

calls. Since our inlining component is not working yet, all calls were inlined by hand. Each program is analysed using the reference input data. Thus, the variables in all READ statements are initialised from the reference data and then treated as compile-time constants.

Table 6.7 presents the experimental results obtained. For a scale of programs such as Applu, *EstimateMisses* obtains close to real miss ratios in about 128 seconds. This translates into a three orders of magnitude speedup over the cache simulator used.

Our results are further discussed below.

Tomcatv from SPECfp95. This example is used to demonstrate the capability of of our method in analysing real codes. The number of iterations of the outermost loop is data-dependent. For the reference input data used, the outermost loop runs for 750 iterations. The only data-dependent IF conditional in the program is always false. The memory accesses contained in this conditional are included in our analysis.

Swim from SPECfp95. This example demonstrates that we can analyse codes consisting of call statements. All calls are parameterless. The outermost loop is an *IF-GOTO* construct, which has been converted into a *DO* statement.

Applu from SPECfp95. This shows that our method is capable of analysing this scale of programs efficiently with a good degree of accuracy. All actual parameters are propagateable. In subroutine SSOR, there are some data-dependent constructs. All but one are guarded by a IF branch that is false at compile time and are thus ignored. The remaining data-dependent IF construct is a WRITE statement for a register-allocated scalar. The memory accesses in this IF conditional are included in our analysis.

Chapter 7

Cache Compiler Optimisations

It is not enough to obtain the cache misses. We need to reason about those misses in order to get better codes that take advantage of the cache. In this chapter we provide a method of using our miss equations for cache-optimising transformations.

```
ALGORITHM:
Supply a population P_0
i=1
while (not finish)
   P_i=Selection(P_{i-1})
   P_i=Reproduce(P_i)
   i=i+1
end
```

Figure 7.1: Simple Genetic Algorithm.

The effectiveness of the memory hierarchy is critical for the performance of current processors. Memory hierarchy behaviour can be enhanced by means of program transformations such as padding, blocking, etc. However, no model has been proposed as an acceptable solution. We need models that can guide transformations in such a way that optimal code is generated.

We present a novel approach that combines genetic algorithms and our miss equations. First, we recall genetic algorithms and how they can be used to optimise functions. Second, we explain how we can parameterise our equations in order to guide different transformations. Finally, we explain how padding can be implemented using our technique.

7.1 Genetic Algorithms

Algorithms for function optimisation are generally limited to convex regular functions. However, there are lots of functions that are not continuous, non differentiable or multi-modal. Stochastic sampling is commonly

used to solve these problems. Whereas traditional search techniques use characteristics of the problem to determine the next sampling point (e.g Gradient), stochastic methods use non-deterministic decision rules [17].

Genetic Algorithms (GAs) are a particular type of stochastic method [31]. They focus on hard problems with objective functions whose properties do not allow using traditional methods. These algorithms search in the solution space of a function simulating the Nature-based process of evolution, that is, the survival of the fittest.

GAs simulate the evolution of a population. Figure 7.1 shows the simplest GA. It starts from a random generated population, and it makes the population evolve by means of basic genetic operators [31] (selection, mutation and crossover) applied to individuals of the current population. The aim of applying genetic operators is producing an improved next generation.

7.1.1 Genetic Algorithm Parameters

Now, we will discuss the practical things regarding the implementation of a GA. We will explain how the individuals may be represented and how the evolution of the population is usually simulated.

Each individual is made up of a set of chromosomes. Usually, each chromosome represents one variable of the function. The fitness of those individuals is computed using the objective function (the one we plan to optimise).

A chromosome representation is needed to represent each individual in the population. Genetic algorithms require the natural parameter set of the optimisation problem to be coded as a finite-length string over some finite alphabet. Although it has been shown that using large alphabets gives better results [52], the most used one is such as {0,1}.

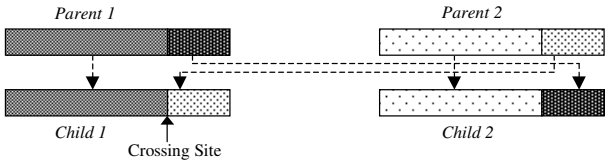


Figure 7.2: Schematic of simple crossover.

Thus, each chromosome is made up of a sequence of genes from a certain alphabet.

Genetic operators provide the basic search mechanism of the GAs, creating new solutions based on the solutions that exist. The *selection* of individuals to produce successive generations plays an extremely important role. A common selection approach assigns probability of selection to each individual depending on its fitness. Individuals with higher fitness have a higher probability of contributing one or more offsprings to the next generation. Then, individuals are selected depending on this probability.

Crossover takes two individuals and produces two new individuals with a given probability, merging the genetic material in a random point (named cross site). In the case they do not crossover, both individuals are added to the new population (see Figure 7.2). *Mutation* changes one individual to produce a new one by flipping some of its genes. Both crossover probability and mutation probability have to be determined empirically, and are related to the size of the population.

The GA must be provided with an initial population (see Figure 7.1) that is created randomly. GAs move from generation to generation, and the usual termination criterion is the number of generations, although

other criteria can be used [31].

7.2 Parametric Miss Equations

When implementing an automated optimisation by means of our miss equations, we should first determine the relationship between the cache behaviour and a set of parameters that describe the actual optimisation. In our case, we may use the number of cache misses yielded by *Estimate Misses* (see Section 5.3). Thereafter, compiler-writers use optimisation techniques that find parameter values that optimise the equations.

Our equations are fully parameterisable. Such parameters range from base addresses to loop bounds. Any parameter can be specified, and once the parameter values are provided, we can get the number of misses calling either *FindMisses* or *EstimateMisses*.

From the best of our knowledge, there exist two implementations of compiler-optimisations following this approach. Abella *et al.* [1] presents a feasible implementation of the blocking technique. In the following section, we will recall how padding can be implemented parameterising the miss equations. For the interested reader, we refer to [68].

7.3 Automatic Near-Optimal Padding

We will use the following terminology:

- C_s is the cache size.
- Var_i is variable number i.
- D_i is the number of dimensions of Var_i.

- dim_{ij} is the size of the dimension j of Var_i.
- S_i is the size of Var_i.
- mem_i is the original base address of Var_i.
- P_Base_i stands for the inter-variable padding between Var_i and Var_{i-1}.
- P_Dim_{ij} is the intra-variable padding applied to dim_{ij}.
- P_S_i is the size of Var_i after padding (see Fig. 7.3).
- We define Δ_i as $P\_S_i - S_i$.

In the following sections, we will first describe the appropriate parameters that define inter-padding (changing variable base address) and intra-padding (increasing array dimension size). Secondly, we will provide the function that relates these parameters to our miss equations, along with experimental results showing the usefulness of this approach.

7.3.1 Inter-variable padding

When inter-variable padding is applied only the base addresses of the variables are changed. Thus, we define for each memory variable Var_i, a variable $P\_Base_i,\ i = 0 \ldots k$ (where k is the number of variables):

$$0 \leq P\_Base_i \leq C_s - 1$$

Note that padding a variable is equivalent to modifying the initial addresses of the other variables (see Figure 7.3). Thus, after padding,

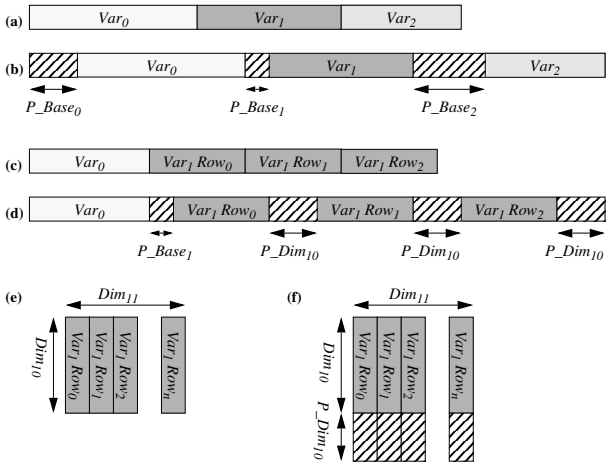


Figure 7.3: Data layout: (a) before inter-variable padding, (b) after inter-variable padding (c) before padding, (d) after padding, (e) 2-D array, (f) 2-D array after intra-variable padding

the memory variable base addresses are computed as follows[1]:

$$BaseAddr(Var_i) = mem_i + \sum_{k=0}^{k \leq i} P\_Base_k$$

7.3.2 Adding intra-variable padding

The result of applying both inter- and intra-variable padding is that all base addresses and sizes of every dimension of each memory variable may change. They are initially set according to the values given by the compiler. For each memory variable $Var_i, i = 0 \ldots k$ we define a set of

[1]We assume variables are stored in memory in the same order as they are numbered.

variables $\{P\_Base_i, P\_Dim_{ij}\}$, $j = 0 \dots D_i$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

After padding, memory variable base addresses are computed in the following way (see Figure 7.3):

$$BaseAddr(Var_i) = mem_i + \\ + \sum_{k=0}^{k<i}(P\_Base_k + \Delta_k) + P\_Base_i$$

and the size of the dimensions are:

$$Dim_i(Var_j) = dim_{ji} + P\_Dim_{ji}$$

7.3.3 Padding Model

We define f as the function that represents the number of misses for each possible value of the padding variables. The expression is as follows:

$$f \quad \longmapsto \quad \#Misses \tag{7.1}$$

$$f : \underbrace{[0, C_s - 1]}_{P\_Base_0} \times \underbrace{[0, C_s - 1]^{D_0}}_{P\_Dim_{0j}} \times \dots \times \underbrace{[0, C_s - 1]}_{P\_Base_k} \times \underbrace{[0, C_s - 1]^{D_k}}_{P\_Dim_{kj}} = \\ = f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0}, \dots, P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

Note that $[0, C_s - 1]^{D_i}$ represents the domain of the different P_Dim_{ij} of the variable Var_i.

Therefore, we can define padding as the following optimisation problem:

$$MIN \quad f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0} \dots, P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

$$i = 0 \dots k$$

where f is called the *objective function*.

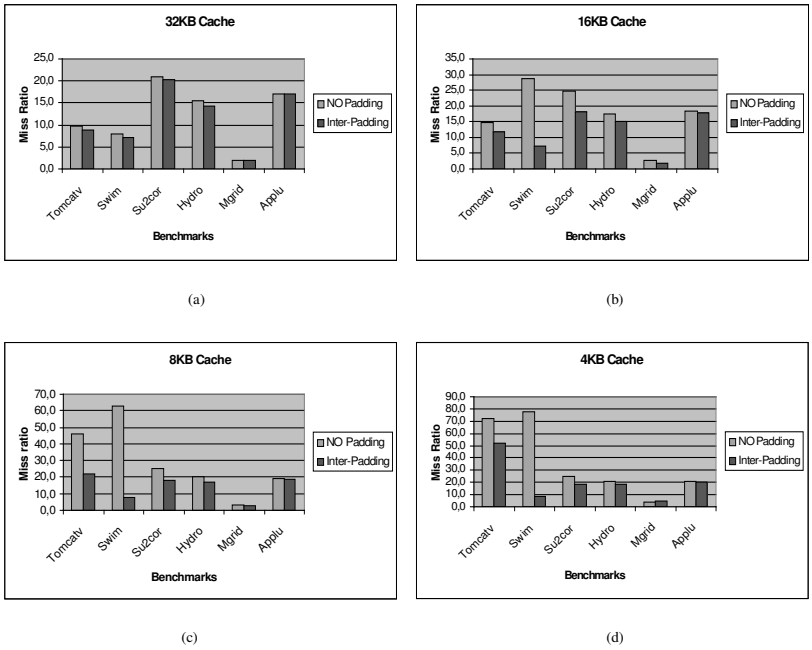


Figure 7.4: Miss ratio before and after inter-variable padding for different cache sizes.

7.3.4 Performance Evaluation for the SPECfp95

SPECfp95 applications have a relatively small working set with respect to current applications. Thus, the results for the smaller cache sizes may be more representative of what we can expect today for larger caches and bigger applications. Two sets of programs can be distinguished:

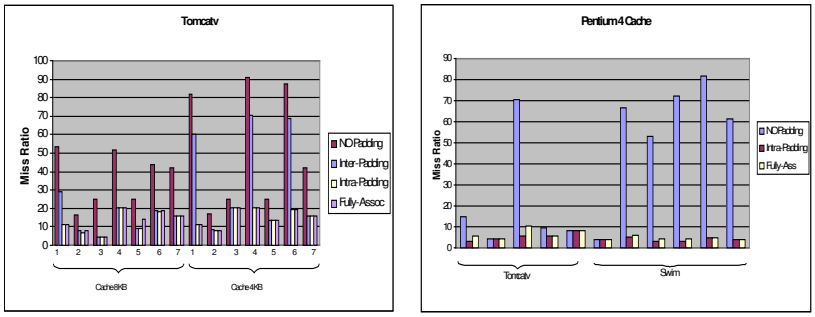
- **Set1** is composed of programs Tomcatv and Swim. The miss ratio of this set of programs is highly affected by cache size. In addition many of the misses are due to conflicts [20].

- **Set2** is composed of programs Su2cor, Hydro, Mgrid, and Applu. The miss ratio of this set of programs is quite insensitive to the cache size. In addition all the programs of this set have practically no conflict misses [20].

Since the objective of padding is to eliminate conflict misses, for **Set2** we obtain a small improvement when applying inter-variable padding due to the low number of conflicts. Su2cor, which is the program with the highest conflict miss ratio in this set, experiences the highest improvement (e.g 27% miss rate reduction for a 16Kbyte cache). In addition, another source of improvement is that the proposed inter-variable padding technique also aligns the data structures with cache lines, which reduces compulsory misses.

On the other hand, inter-variable padding provides a huge improvement in miss ratio for **Set1**. Note that for both programs, a small improvement is obtained for a 32Kbyte cache (Figure 7.4.a). This is caused by the fact that almost no conflicts arise for 32Kbyte caches or bigger for these programs due to the relatively small working set of the SPECfp95 applications. However, the smaller the cache the bigger the miss ratio and the bigger the improvement that inter-variable padding obtains.

For the Tomcatv program, the miss ratio also grows significantly when the cache size is reduced (9.5%, 14.8%, 46.0%, and 72.1% respectively for the different cache sizes). However, the miss ratio after inter-variable padding varies significantly with the cache size (8.8%, 11.8%, 21.6%, and 52%). This variation is caused by capacity misses that grow when the cache is reduced, and by intra-variable conflict misses (e.g., conflicts among distinct rows and columns of the same array) whose frequency also grows when the cache is reduced. Inter-variable padding



(a) (b)

Figure 7.5: (a) Miss ratio for different Tomcatv loop nests before and after inter- and intra-variable padding (b) Miss ratio for the Tomcatv and Swim loop nests for the Pentium 4 L1 cache.

does not remove the latter type of conflicts, which are the target of intra-variable padding.

Figure 7.5.a shows the miss ratio for the different loop nests of the Tomcatv program. The figure shows the miss ratio for each loop after applying inter- and intra-variable padding. It also shows the miss ratio before padding and that of a fully-associative. As we observed before, inter-variable padding does not remove all conflicts misses because there are intra-conflict misses. Intra-variable padding achieves about the same miss rate as the fully-associative cache, which means that the proposed padding algorithm removes practically all conflict misses.

Chapter 8

Related Work

This chapter discusses the different related work. We present other tools that try, in some way or another, to analyse cache memory behaviour. We give some details for those analytical approaches that we believe are closer to our analysis.

Programs must exhibit sufficient locality to achieve good cache performance. Compiler optimisations for improving the cache behaviour need to have detailed knowledge about the number and causes of cache misses. Such an information can be obtained by time-consuming cache simulation [66] and architecture-dependent hardware counters [4].

Memory simulation techniques are very accurate, flexible and can provide rich information. They are usually based on trace-driven simulation [40, 30, 53, 62, 25, 49, 32, 7, 51, 67]. However, these techniques are very slow (usually several orders of magnitude). For instance, the slowdown exhibited by all simulators surveyed in [66] is in the range of 45-6250. There are some innovative methods that have been proposed with the objective of reducing the exhibited slowdown [50, 46, 75]. Nevertheless, these methods provide little information (usually only miss ratio), that is, they trade information for speed.

There are other tools based on hardware counters (e.g., [4]) provided by some microprocessors. These tools are fast and accurate. They have no flexibility since they can only be used to analyse the memory architecture of the actual microprocessor. In addition, they provide a limited set of results depending on the particular counters provided by a particular machine. Information like conflict misses between two particular memory references cannot be obtained with current hardware counters.

Analytical methods use mathematical formulas to provide a characterisation of a program's cache behaviour so that we can not only obtain the number of cache misses but also reason about the causes of such misses from these formulas. The ultimate goal is to develop an analytical method that can provide accurate assessments of when and why cache misses occur using a reasonable amount of computational resources (e.g., CPU time, memory and disk usage). Then such a method will be

useful in guiding various automatic memory optimisations and also in improving the simulation times of cache simulators and profilers.

Some static analysis techniques [63] have limited accuracy, due to unknown information at compile time. For instance, unknown loop bounds or unknown initial addresses of data structures can degrade the accuracy of the results.

A solution to this problem is to use hybrid techniques (which combine the very best from both approaches), such as SPLAT [60]. SPLAT is a static analysis technique improved with some profile (dynamic) information. This hybrid technique is fast, flexible and can provide many different information like other static techniques. In addition it is accurate because the information unknown at compile time is provided by a profiling. Due to simplifications in the analysis, SPLAT is not capable of analysing interferences in applications with complex interference patterns and can only analyse direct mapped caches.

Porterfield [54] introduces the concept of overflow iteration for predicting the miss ratio for a fully set-associative LRU cache. Ferrante, Sarkar and Thrash [21] provide closed-form formulas to estimate the capacity misses of a loop nest. Temam, Fricker and Jalby [63] also consider conflict misses but for a subset of array references studied in this paper. Wolf and Lam [77] propose to use vectors to describe data reuse for uniformly generated references in a perfect loop nest. They also use reuse vectors to derive an estimate of cache misses to guide their data locality algorithm. Xue and Huang [82] report an improvement. Gannon *et al* [24] and Wolfe [79] discuss the use of reference window for predicting cache misses.

Recently, Weikle *et al* [72] introduce a trace-based idea of viewing caches as filters. Their framework can potentially handle any programs

N	BJ	BK	C_s	L_s	k	Δ_P	Δ_E
200	100	100	16	8	2	6.23	0.1
200	100	100	256	16	2	2.73	0.5
200	200	100	32	8	1	6.88	0.06
200	200	100	128	8	2	2.86	0.05
200	200	100	128	32	2	44.25	16
200	50	200	16	4	1	4.62	0.05
200	100	200	32	8	2	12.51	0.1
200	100	200	64	16	1	3.31	0.4
400	100	100	16	8	2	4.48	0.03
400	100	100	256	16	2	4.26	0.5
400	200	100	32	8	1	2.65	0.4
400	200	100	128	8	2	5.82	0.05
400	200	100	128	32	2	44.68	16
400	50	200	16	4	1	2.02	0.05
400	100	200	32	8	2	5.55	0.06
400	100	200	64	16	1	7.12	0.3

Table 8.1: Comparison with Fraguela et al's probabilistic method using MMT. Δ_p denotes the relative error between the estimated and real miss ratios for the probabilistic method and Δ_E for our *EstimateMisses*.

consisting of any pattern of memory references.

8.1 Analytical Methods

We review in detail the three recent compile-time analytical methods for predicting cache behaviour [13, 22, 28].

Ghosh *et al* [28] present their seminal work on using the CMEs to analyse statically a program's cache behaviour. This framework is targeted at isolated perfect loop nests, consisting of straight-line assign-

ments, by emploiting only the reuse vectors between uniformly generated references in the same nest. They show that the CMEs can provide insights in choosing appropriate tile and padding sizes. Since analysing all iteration points is costly, an efficient implementation of the CMEs based on polyhedral theory and statistical sampling techniques is discussed in [10, 69].

Fraguela *et al* [22] rely on a probabilistic analytical method to provide a fast estimation of cache misses. While allowing multiple nests, they exploit only the reuse between references contained in the same nest (as can also be done in the CMEs.) These references differ by constants in their matching dimensions, forming a subset of uniformly generated references considered in the CMEs. Their experimental results using three examples indicate that their method can achieve a good degree of accuracy in estimating cache misses for perfect nests.

Their two perfect nest examples can be analysed by the CMEs and are not compared here. The other one is a 3-D blocked imperfect nest computing AB^T (named MMT in Figure B.2). Table 8.1 compares their method with ours. Our *EstimateMisses* produces better results in all cases. The two largest relative errors occur since the total number of misses is small.

Chatterjee *et al* [13] present an ambitious method for *exactly* modelling the cache behaviour of loop nests. They use Presburger formulas to specify a program's cache misses, the Omega Calculator [55] to simplify the formulas, PolyLib [74] to obtain an indiscriminating union of polytopes, and finally, Ehrhart polynomials to count the number of integer points (i.e., misses) in each polytope [14]. They can formulate Presburger formulas for a looping structure consisting of imperfect nests, IF statements, references with affine accesses and non-linear data layouts.

That is, they are not restricted to uniformly generated references and linear array layouts. When solving their formulas, they provide only the cache miss numbers for 20×20 and 21×21 matrix multiplication without giving any execution times. In the case of matrix-vector product, they give Presburger formulas for $N = 100$ but do not solve them.

Exact analysis is undoubtedly useful but can be too costly for realistic codes to be of any use in guiding compiler optimisations to improve performance. *FindMisses* can be exact if all necessary reuse vectors are used. Our current implementation exploits only the reuse among uniformly generated references. One future work is to derive systematically the reuse vectors for non-uniformly generated references.

Neither of the three methods discussed above can handle call statements. In comparison with these existing techniques, our method can analyse complete regular programs efficiently with high accuracy.

8.2 Cache Compiler Optimisations

Caches improve the speed of programs by reducing the number of accesses to the slow upper levels of the memory hierarchy. Conflict misses may represent the majority of intra-nest misses and about half of all cache misses for typical programs and cache architectures [51].

Many hardware techniques have been proposed to reduce conflict misses, such as the victim cache [38] or pseudo-random placement functions [65]. Software techniques are attractive because they do not increase the hardware complexity and may be very effective for regular programs where the compiler can perform an accurate locality analysis. Moreover, they can complement hardware techniques.

Among software techniques to avoid self-interferences we can point

out the tile size selection proposed by Lam *et al.* [44]. Coleman and McKinley [15] improved that technique by allowing rectangular tiles. Temam *et al.* [64] proposed to use a buffer where the data to be manipulated is copied. Unfortunately, the copy operation itself causes cache conflicts and has some overhead.

There are many other proposals to transform the order in which the iteration space is traversed, such as loop interchange, loop permutation, loop distribution, etc. [78, 12, 41]. Although these transformations may affect the number of conflict misses, they are not specially targeted to minimise them.

Some padding techniques have been previously proposed by other authors. Rivera and Tseng [58, 59] propose several simple heuristics that are addressed to eliminate conflicts in some particular cases. They mainly focus on conflicts that occur on every loop iteration, and in some assumptions about the effect of array column sizes and distance among starting addresses of variables on the conflict miss ratio. These heuristics may be more or less effective depending on the particular reference pattern of each program.

On the other hand, Ghosh *et al.* [27] propose a padding technique based on using the Cache Miss Equations for conflicting arrays that have the same column size. Their technique finds the optimal padding if there is a padding such that the total number of replacement misses after padding is zero. However, if such a padding does not exist, their technique does not provide any solution. Note that replacement misses include both conflict and capacity misses and one may expect the case where replacement misses cannot be decreased up to zero to be common. In their experiments, this only happens for one out of the seven loops examined but most of their benchmarks are small kernels.

Chapter 9

Conclusions

We wrap up our work. We make the conclusions of the current work, stressing the goals achieved. Finally, we present open issues that may be tackled in the future.

We have introduced a new characterisation of reuse vectors for quantifying reuse across multiple nests. Based on these reuse vectors, we have developed an analytical method for statically predicting the cache behaviour of complete programs with regular computations. We outlined two algorithms for computing cache misses. *FindMisses* analyses all iteration points and can predict exactly the cache misses for programs of small problem sizes. *EstimateMisses* analyses a sample of all memory accesses and can achieve close to real cache miss ratios in practical cases efficiently.

We can analyse IF statements with compile-time-analysable conditionals. In the presence of these conditionals, different references may be executed in different parts of iteration spaces, which are not necessarily convex. We described how reuse vectors are calculated and how the miss equations are formed and solved. Our replacement miss equations are formulated and solved by taking into account the fact that the RISs for different references can be different.

The experimental results obtained for three kernels and three whole programs (one of which is Applu from SPECfp95 with 3868 lines, 16 subroutines and 2565 references) confirm the efficiency and accuracy of our method. Our method can be used to guide compiler optimisations and improve the speeds of cache simulators.

We have also shown how our method can guide cache-compiler optimisations. Combining the equations with genetic algorithms allows performing near-optimal optimisations such as padding.

9.1 Future Work

This thesis is carried out in a project group (WCET group). The activities mainly concern issues pertaining to program flow and program analysis, low-level simulation and analysis. It also aims to provide accurate program flow-based information about cache behaviour that will aid the cache part of the low-level analysis.

The overall goal of the WCET project is to create an automated tool for WCET analysis of realistic codes in real-time systems. It aims at "real" programs as coded for embedded real-time systems. For this reason, we will analyse full C (i.e., pointers, recursion, and unstructured code using jumps). We plan to rely on manual annotations where our automatic approach is insufficient.

Activities have been divided into two sub-projects. Firstly, an automated program flow analysis is indispensable. The work presented focuses on the second part: in a later stage, the tool should be able to analyse systems with caches with reasonable precision.

While this work represents a useful step towards a mechanical analysis of complex program constructs, there are several important constructs that are still non-analysable, including data-dependent conditionals and pointers. We are presently working on developing an analytical method for their efficient analysis. This includes:

Constructs Data-dependent constructs such as variable bounds, data-dependent IF conditionals and indirection arrays are still not analysable. We plan to investigate techniques for their analysis. To go beyond FORTRAN, we need to cope with pointers and recursive calls. We plan to generalise our analytical method to the full C language.

Reuse Vectors The accuracy of our method strongly depends on

the computation of reuse vectors. We plan to extend the basic reuse vectors analysis, in order to express basic reuse among non-uniformly generated references.

We intend to investigate benefits and limitations of these challenging but important research directions.

Appendix A

Background

This chapter explains some terminology that is often used in this dissertation. We first introduce memory hierarchy and describe different kind of cache memories. Secondly, we give some details regarding locality analysis, focusing on the reuse vectors methodology.

A.1 Memory Hierarchy

Many researchers call the memory system the bottleneck of current high-performance computers, which with interconnection it is going to be the main topic of research in 2010. The memory system limits how fast data is brought to the processor and also how fast can data be received from the processor.

The gap between main memories and processors performance is increasing every year. Whereas processor has improved about 60% per year, memory access time has only decreased at less than 10% per year. This large latency, which is increasing at a rate of 45% per year, is a primary obstacle to improve general system performance. Memory hierarchy and cache memories were introduced as a hardware solution to hide this gap, becoming more and more significant with the widening performance difference.

Memory hierarchy consists on stowing different memories between the processor and the main memory. The closer the memory is to the processor, the faster it is. Since the caches are smaller than the main memory, they can be designed using faster and more expensive techniques.

The characteristic that drove the invention of cache memories, and that make them work well in general is called *locality*:

Rule 90/10 90% of the memory accesses are done only by 10% of the instructions.

Locality There exists temporal locality (i.e., it is very likely that instructions access data that has been already fetched), and spatial locality (i.e., it is very likely that instructions access data nearby data that has been already fetched).

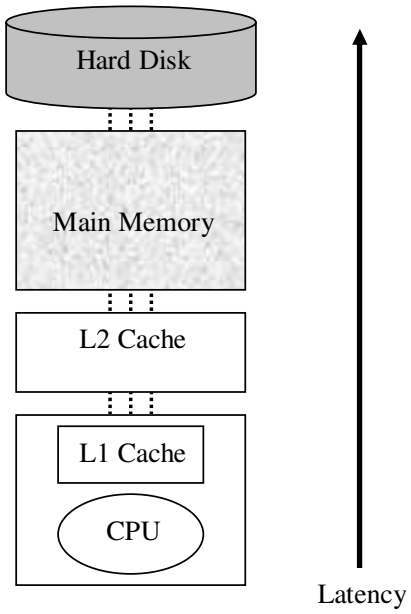


Figure A.1: Memory hierarchy.

Figure A.1 shows a common memory hierarchy scheme. Memory hierarchy aims at having data that it is supposed to be accessed very often in memory levels very close to the processor. Since these memories are faster, the latency is smaller. Thus, the general performance is increased.

A.1.1 Cache Memories

Definition 1.1 *Cache memory* is a very fast and small memory which contains a small subset of the data stored in a larger (and slower) memory (see Figure A.2).

Cache memories try to capture the most frequently accessed data items.

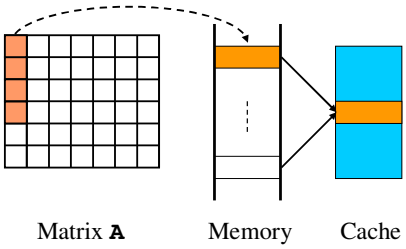


Figure A.2: How data is stored in both main memory and cache.

Placed between processor and main memory, it bridges queries from the processor. When a memory request is generated, the request is first presented to the cache memory, and only in the case it cannot respond is presented to the main memory. This can be summarised as follows:

1. If the cache has the data that the processor needs (*cache hit*), it brings the data to the processor.

2. Otherwise, we suffer a *cache miss* and the main memory must be accessed.

Let us look at the various parameters that affect to cache design. We explain in the following sections how data is organised in the cache and different policies that keep the coherence between main memories and caches. The interested reader is pointed to [35].

A.1.2 Cache Organisation

Caches are characterised by the following parameters. *Cache size* (C_s) defines the total number of bytes it has. *Line* or *block* is the data item that is delivered between main memory and cache memory. The *line size* (L_s) determines how many contiguous bytes are fetched from memory.

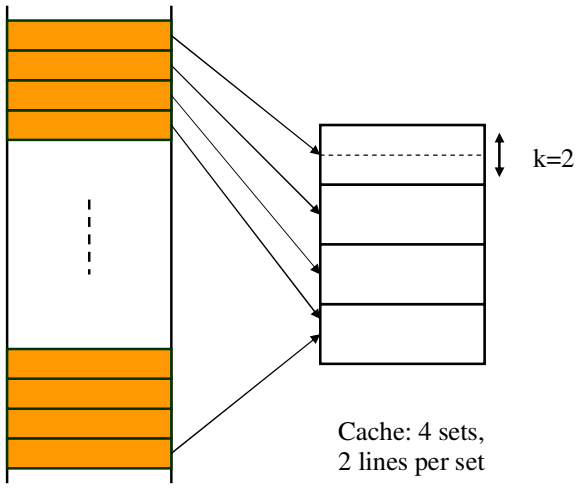


Figure A.3: Mapping of such a 2-way associative cache.

Cache Set is a collection of cache lines. *Associativity (k)* is the number of cache lines in a cache set. Depending on the location in the cache a block from memory can reside in, we have different cache organisations.

Direct-Mapped Given a block, it can only reside in one exact location. The cache size is

$$C_s = L_s \times Number\_Lines$$

k-way set associative Each memory block may reside in exactly k locations. Figure A.3 shows a 2-way set associative cache, with 4 sets.

Fully associative A block from memory can reside in any location in the cache.

Two factors are critical at the moment of choosing an index scheme that maps memory lines onto cache lines (sets); firstly, the chosen function should have a cheap and easy implementation in hardware, and secondly it is important that it has a good behaviour on any kind of regular address patterns. Modulo function (modulo the number of cache sets) is the most common index function used. However, alternative schemes may help programs to take advantage of cache memories. Prime-modulus functions [45] and skewing functions [36] have been tested successfully. The use of XOR functions was proposed by Frailong *et al.* [23], and some pseudo-random placements by Rau *et al.* [57].

Let Mem_Addr be the memory address. We have:

$$\begin{aligned} Mem\_Line &= \lfloor Mem\_Addr/L_s \rfloor \\ Cache\_Set &= Mem\_Line \bmod \mathcal{N} \end{aligned}$$

where $\mathcal{N} = C_s/k$ is the number of cache sets.

A.1.3 Replacement Policies

For those operations that result in a cache miss, the item is retrieved from the main memory and copied into the cache, resulting in some other item removed from the cache to make room for this new item. The cache replacement policy (i.e., which item we remove from the cache) is crucial for performance.

The following expression gives the average execution time of one memory access:

$$t_{eff} = ht_{hit} + (1-h)t_{miss}$$

where h is the probability of having a hit (known as *hit ratio*). Let be the main memory ten times slower than the cache. If $t_{hit} = x$, then

$t_{miss} = x + 10x$. Then, a decrease in the hit ratio of 0.01, results in a roughly increase of t_{eff} of ten-percent.

Nearly all caches in commercial products have least-recently used (LRU) replacement policy to manage the different lines we have in a set. Other algorithms are:

Random One line is chosen randomly.

FIFO The line that was brought in the first time is replaced (even though it may happen that it is accessed very often).

A.1.4 Writing to the Cache

Special actions should be taken in case of WRITE operations. 10-30% of the memory accesses are WRITEs. Handling them is somewhat tricky because of the interaction of the cache with input/output systems. Keeping cache coherence with the main memory is very important. Different solutions have been presented, depending on whether there exists a cache miss or a cache hit.

When having a cache hit:

Write through We write the modified item both in the cache and in the memory. Since main memory is rather slow, hardware solutions like buffering are implemented in order to speed up the whole process.

Copy back Data is written in the main memory only when the cache line is replaced. Thus, the memory does not contain updated information.

In case of suffering a cache miss:

Write allocate The memory line is modified and it is brought to the cache afterwards.

No write allocate The memory line is modified, but the data is not brought to the cache.

When specifying a writing policy, both hit and miss policies should be provided. The two most common configurations are:

- Copy back with Write allocate.
- Write through with No write allocate.

A.2 Locality Analysis

Cache memories usually present very low associativity, which may result in data being replaced before it is reused. Furthermore, there may exist programs without sufficient locality in their access, which makes them spend a lot of time transferring data between main memory and cache. Improving program performance requires a clear knowledge of the reasons behind its behaviour. Locality analysis brings this information that can be further used by either programmers or automatic compiler optimisations to tune the code in order to achieve better cache performance.

A.2.1 Terminology

This section introduces some terminology that is later used.

Definition 1.2 *Iteration point* is each one of the different iterations of a loop nest.

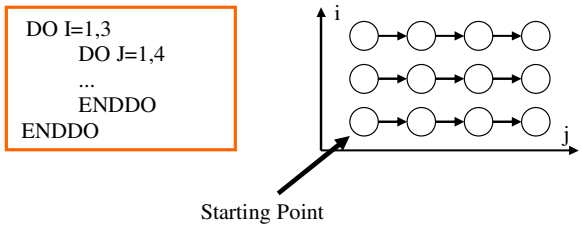


Figure A.4: Example of loop nest and its iteration space.

Definition 1.3 *Iteration space* is the collection of all the iteration points.

Figure A.4 shows one loop nest and its iteration space.

We present the following definition to avoid any confusion regarding memory accesses.

Definition 1.4 *Reference (memory reference)* is a static read or write. This reference will probably result in a dynamic memory access.

Our approach deals with both scalars and array accesses. The memory address of such an access is given by the following expression. Let $R_A = A[f_{d_A-1}, \ldots, f_1, f_0]$ (d_A stands for the number of dimensions of A) be a reference to array variable A:

$$@R_A[f_{d_a-1}, \ldots, f_1, f_0] = offset + f_0 + \sum_{k=1}^{d_A-1}(size\_dim_{k-1} \times f_k)$$

where $size\_dim_{k-1}$ is the size of array A under dimension k. Notice that we have used a column-major data layout, which is the one assumed for FORTRAN codes.

A.2.2 Reuse Vectors

Reuse vectors are a mechanism to summarise memory access patterns for loop oriented codes. They were introduced by Wolf and Lam [77]. If

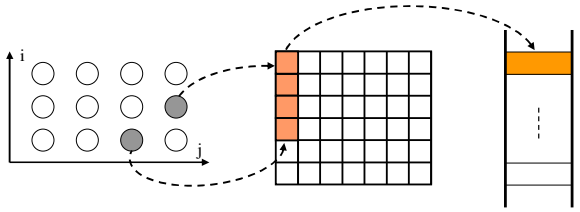


Figure A.5: Example of locality described by means of a reuse vector.

we describe the iteration space as a subset of $\mathbb{Z}^n$, we define the following terms:

Definition 1.5 If a reference accesses the same memory line in two different iteration points $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_2 \succeq \vec{i}_1$, we say there exists a reuse vector $\vec{r}$ (see Figure A.5) and we define it as

$$\vec{r} = \vec{i}_2 - \vec{i}_1$$

We would like to point out the difference between *potential* reuse and *actual* reuse. We say one item is actually reused when we enjoy a hit accessing this data in different iteration points. The existence of reuse does not imply that we enjoy a hit, since the cache line may have been flushed out by another access before the reuse can be realised.

A.2.3 Different Reuse for Different Locality

Given a reference, we classify the reuse into four different groups [77]:

Self Temporal A reference reuses a memory line by accessing the same data in different iteration points.

Self Spatial A reference reuses a memory line by accessing data different from the one accessed before.

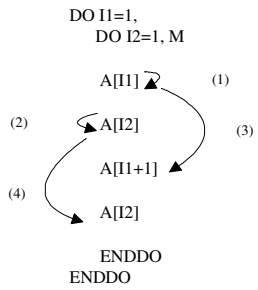


Figure A.6: Example of reuse

Group temporal A reference accesses the same memory line as another reference by accessing exactly the same item.

Group Spatial A reference accesses the same memory line as another reference by accessing different items contained in the same memory line accessed earlier.

If we take a closer look to these definitions, we can see that temporal reuse is a subset of spatial reuse. Whereas the temporal locality can be exploited as many times as we want, the possibility of exploiting spatial locality depends on the L_s.

A.2.4 Example

We consider the code shown in Figure A.6. We show four different reuse vectors that illustrate the previous statements.

Vector 1 Reference A[I1] has self temporal reuse in the innermost loop

nest, since it is accessing the same elements for iterations (I1, I2), $1 \leq I2 \leq M$.

Vector 2 Reference A[I2] has self spatial reuse in the innermost loop nest, since it is accessing the same memory line L_s times (assuming data is aligned to the cache line).

Vector 3 This vector shows group spatial reuse since both references are accessing the same cache line in the innermost loop nest.

Vector 4 Finally, this vector gives an example of group temporal reuse. Both references are accessing, in all iteration points, the same data.

Appendix B

Codes

We show the different large codes we have used to validate our approach.

The codes come from different benchmarks used to evaluate the performance of supercomputers. They consist of kernel-based applications, which implement very intensive regular computations, such as Swim (it solves the system of shallow water equations) or Applu (which computes parabolic / elliptic partial differential equations).

Our benchmarks contain programs from Livermore, Linpack, Lapack, SpecFp95 and Perfect Benchmarks.

```
PROGRAM LU
PARAMETER (N = 100)
REAL*8 a(N,N)
DO i = 1,N
 DO j = i+1,N
  a(j,i) = a(j,i)/a(i,i)
  DO k = i+1,N
   a(j,k) = a(j,k)-a(j,i)*a(i,k)
  ENDDO
 ENDDO
ENDDO
END
```

```
...
DO i = 1,N
 DO j = i+1,N
  DO k = i+1,N
   IF (k .EQ. i+1) THEN
    a(j,i) = a(j,i)/a(i,i)
   ENDIF
   a(j,k) = a(j,k)-a(j,i)*a(i,k)
  ENDDO
 ENDDO
ENDDO
END
```

```
PROGRAM MM
PARAMETER (N=100)
REAL*8 a(N,N), b(N,N), c(N,N)
DO i = 1,N
 DO j = 1,N
  a(i,j) = 0
  DO k = 1,N
   a(i,j) = a(i,j)+b(i,k)*c(k,j)
  ENDDO
 ENDDO
ENDDO
END
```

```
...
DO i = 1,N
 DO j = 1,N
  DO k = 1,N
   IF (k.EQ.1) THEN
    a(i,j) = 0
   ENDIF
   a(i,j) = a(i,j)+b(i,k)*c(k,j)
  ENDDO
 ENDDO
ENDDO
END
```

```
PROGRAM LWSI
PARAMETER (NS = 10, natoms = 100)
DOUBLE PRECISION xt, yt, xc, yc, zc
DOUBLE PRECISION zero, wsin, wcos, z, xs
DIMENSION xc(natoms, ns), yc(natoms, ns)
DIMENSION zc (natoms, ns), xt (natoms)
DIMENSION wsin(1), wcos(1), zero(1), z(1)
DIMENSION xs(1), yt (natoms)
DO i = 1, ns, 1
 xt(1) = xt(2)+wcos(1)
 xt(3) = xt(1)
 yt(2) = zero(1)
 DO j = 1, ns, 1
  yt(1) = yt(2)+wsin(1)
  yt(3) = yt(2)-wsin(1)
  z(1) = zero(1)
  DO k = 1, ns, 1
   DO l = 1, natoms, 1
    xc(l,k) = xt(l)
    yc(l,k) = yt(l)
    zc(l,k) = z(1)
   ENDDO
   z(1) = z(1)+xs(1)
  ENDDO
  yt(2) = yt(2)+xs(1)
 ENDDO
 xt(2) = xt(2)+xs(1)
ENDDO
END
```

```
...
DO i = 1, ns, 1
 DO j = 1, ns, 1
  DO k = 1, ns, 1
   DO l = 1, natoms, 1
    IF (j.EQ.1 .AND. k.EQ.1
     .AND. l.EQ.1) THEN
     xt(1) = xt(2)+wcos(1)
     xt(3) = xt(1)
     yt(2) = zero(1)
    ENDIF
    IF (k.EQ.1 .AND. l.EQ.1) THEN
     yt(1) = yt(2)+wsin(1)
     yt(3) = yt(2)-wsin(1)
     z(1) = zero(1)
    ENDIF
    xc(l,k) = xt(l)
    yc(l,k) = yt(l)
    zc(l,k) = z(1)
    IF (l.EQ.natoms) THEN
     z(1) = z(1)+xs(1)
    ENDIF
    IF (k.EQ.ns .AND. l.EQ.natoms) THEN
     yt(2) = yt(2)+xs(1)
    ENDIF
    IF (j.EQ.ns .AND. k.EQ.ns
     .AND. l.EQ.natoms) THEN
     xt(2) = xt(2)+xs(1)
    ENDIF
   ENDDO
  ENDDO
 ENDDO
ENDDO
END
```

Figure B.1: Three examples (with original and transformed programs): LU (without pivoting) is taken from Lapack, LWSI is a 4-D imperfect loop nest from LWSI and MM is from Livermore kernels.

```
PROGRAM Hydro
REAL*8 ZA, ZP, ZQ, ZR, ZM, ZB, ZU, ZV, ZZ
DIMENSION ZA(jn+1,kn+1), ZP(jn+1,kn+1), ZQ(jn+1,kn+1)
DIMENSION ZR(jn+1,kn+1), ZM(jn+1,kn+1)), ZB(jn+1,kn+1)
DIMENSION ZU(jn+1,kn+1), ZV(jn+1,kn+1), ZZ(jn+1,kn+1)
T= 0.003700D0
S=0.004100D0
DO k= 2,KN
  DO j= 2,JN
    ZA(j,k)=(ZP(j-1,k+1)+ZQ(j-1,k+1)-ZP(j-1,k)-ZQ(j-1,k))*(ZR(j,k)
            +ZR(j-1,k))/(ZM(j-1,k)+ZM(j-1,k+1))
    ZB(j,k)= (ZP(j-1,k)+ZQ(j-1,k)-ZP(j,k)-ZQ(j,k))*(ZR(j,k)
            +ZR(j,k-1))/(ZM(j,k)+ZM(j-1,k))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZU(j,k)= ZU(j,k)+S*(ZA(j,k)*(ZZ(j,k)-ZZ(j+1,k))-ZA(j-1,k)*(ZZ(j,k)-ZZ(j-1,k))
            -ZB(j,k)*(ZZ(j,k)-ZZ(j,k-1))+ZB(j,k+1) *(ZZ(j,k)-ZZ(j,k+1)))
    ZV(j,k)= ZV(j,k)+S*(ZA(j,k)*(ZR(j,k)-ZR(j+1,k))-ZA(j-1,k) *(ZR(j,k)-ZR(j-1,k))
            -ZB(j,k) *(ZR(j,k)-ZR(j,k-1))+ZB(j,k+1) *(ZR(j,k)-ZR(j,k+1)))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZR(j,k)= ZR(j,k)+T*ZU(j,k)
    ZZ(j,k)= ZZ(j,k)+T*ZV(j,k)
  ENDDO
ENDDO
END
```

Figure B.2: Three kernels.

```
PROGRAM MGRID
REAL*8 U,Z
DIMENSION U(M,M,M), Z(M,M,M)
DO 400 I3=2,M-1
  DO 200 I2=2,M-1
    DO 100 I1=2,M-1
      U(2*I1-1,2*I2-1,2*I3-1)=U(2*I1-1,2*I2-1,2*I3-1)
        +Z(I1,I2,I3)
100 CONTINUE
    DO 200 I1=2,M-1
      U(2*I1-2,2*I2-1,2*I3-1)=U(2*I1-2,2*I2-1,2*I3-1)
        +0.5D0*(Z(I1-1,I2,I3)+Z(I1,I2,I3))
200 CONTINUE
  DO 400 I2=2,M-1
    DO 300 I1=2,M-1
      U(2*I1-1,2*I2-2,2*I3-1)=U(2*I1-1,2*I2-2,2*I3-1)
        +0.5D0*(Z(I1,I2-1,I3)+Z(I1,I2,I3))
300 CONTINUE
    DO 400 I1=2,M-1
      U(2*I1-2,2*I2-2,2*I3-1)=U(2*I1-2,2*I2-2,2*I3-1)
        +0.25D0*(Z(I1-1,I2-1,I3)+Z(I1-1,I2,I3)
        +Z(I1, I2-1,I3)+Z(I1, I2,I3))
400 CONTINUE
STOP
```

Three kernels (cont'd)

```
PROGRAM MMT
REAL*8 A, B, D, WB
DIMENSION A(N,N), B(N,N), D(N,N), WB(N.N)
DO J2 = 1,N,BJ
  DO K2 = 1,N,BK
    DO J=J2,J2+BJ-1
      DO K=K2,K2+BK-1
        WB(J-J2+1,K-K2+1)=B(K,J)
      ENDDO
    ENDDO
    DO I = 1,N
      DO K=K2,K2+BK-1
        RA=A(I,K)
        DO J=J2,J2+BJ-1
          D(I,J)=D(I,J)+
              WB(J-J2+1,K-K2+1)*RA
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
END
```

Three kernels (cont'd)

Bibliography

[1] J. Abella, A. González, J. Llosa, and X. Vera. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of 31st International Conference on Parallel Processing (ICPP02) Workshops*, Aug. 2002.

[2] W. Abu-Sufah. *Improving the performance of virtual memory computers.* PhD thesis, University of Illinois at Urbana-Champaign, Nov. 1978.

[3] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behaviour prediction by abstract interpretation. In *Proceedings of Static Analysis Symposium (SAS'96)*, Lecture Notes in Computer Science (LNCS) 1145, pages 52–66. Springer-Verlag, September 1996.

[4] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 85–96, 1997.

[5] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 172–181, 1994.

[6] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: a tool for research on ilp. In *Proceedings of Supercomputing (SC'96)*, 1996. Research Exhibit "Polaris at Work".

[7] R. Bedichek. Talismam: Fast and accurate multicomputer simulation. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95)*, pages 14–24, May 1995.

[8] N. Bermudo and X. Vera. Coyote project: Documentation. Technical Report MRTC Report 39/2001, Mälardalens Högskola, Oct. 2001.

[9] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[10] N. Bermudo, X. Vera, A. González, and J. Llosa. Optimizing cache miss equations polyhedra. In *4th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-4)*, 2000.

[11] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing (SC'92)*, pages 114–124, Minneapolis, Minn., Nov. 1992.

[12] S. Carr, K.S. McKinley, and C-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of VI Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 252–262, Oct. 1994.

[13] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 286–297, 2001.

[14] P. Clauss. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *Proceedings of ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, Philadelphia, 1996.

[15] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, Jun. 1995.

[16] M. H. DeGroot. *Probability and statistics*. Addison-Wesley, 1998.

[17] Y. Ermoliev and R. J.-B. Wets. *Numerical Techniques for Stochastic Optimization*. Springer-Verlag, 1988.

[18] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[19] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17:131–181, 1999.

[20] A. Fernández. A quantitative analysis of the SPECfp95. Technical Report UPC-DAC-1999-12, Universitat Politècnica de Catalunya, March 1999.

[21] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *4th Workshop on Languages and Compilers for Parallel Computing (LCPC'91)*, pages 328–343, 1991.

[22] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[23] J. Frailong, W. Jalby, and J. Lenfant. XOR-schemes: a flexible data organization in parallel memories. In *Proceedings of International Conference on Parallel Processing (ICPP'85)*, pages 276–283, 1985.

[24] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[25] J. Gee, M. Hill, D. Pnevmatikatos, and A.J. Smith. Cache performance of the spec92 benchmark suite. *IEEE Micro*, pages 17–27, Aug. 1993.

[26] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of International Conference on Supercomputing (ICS'97)*, pages 317–324, Vienna, 1997.

[27] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 228–239, 1998.

[28] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[29] S. Ghosh, M. Martonosi, and S. Malik. Automated cache optimizations using CME driven diagnosis. In *Proceedings of International Conference on Supercomputing (ICS'00)*, pages 316–326, 2000.

[30] A.J. Goldberg and J. Hennessy. Performance debugging shared memory multiprocessor programs with mtool. In *Proceedings of Supercomputing (SC'91)*, pages 481–490, 1991.

[31] D. E. Goldberg. *Genetic algorithms in search, optimizations and machine learning.* Addison-Wesley, 1989.

[32] S. Goldschmidt and J. Hennessy. The accuracy of trace-driven simulation of multiprocessors. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*, pages 146–157, May 1993.

[33] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *1993 Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, pages 567–585, Portland, Ore., Aug. 1993. Springer Verlag.

[34] C. A. Healey, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 288–297, 1995.

[35] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufman Publishers, 1996.

[36] D. T. Harper III and J. R. Jump. Vector access performance in parallel memories. *IEEE Transactions on Computers*, C(36):1440–1449, 1987.

[37] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.

[38] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of 17th International Symposium on Computer Architectures (ISCA'90)*, 1990.

[39] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of International Conference on Microprogramming and Microarchitecture*, pages 285–296, 1998.

[40] K. Kennedy, D. Callahan, and A. Porterfield. Analyzing and visualizing performance of memory hierarchy. In *Instrumentation for Visualization*. ACM Press, New York, 1990.

[41] K. Kennedy and K.S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Technical Report COMP TR92-189, Rice University, August 1992.

[42] S. K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, 1996.

[43] I. Kodukul, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 346–357, Las Vegas,NA, 1997.

[44] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proceedings of IV International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, Apr. 1991.

[45] D. Lawrie and C. Vora. The prime memory system for array access. *IEEE Trans. Computers*, C(31):435–442, 1982.

[46] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, Oct. 1994.

[47] Y. T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 298–307, 1995.

[48] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 97–108, 1994.

[49] P. Magnusson. A design for efficient simulation of a multiprocessor. In *Proceedings of the Western Simulation Multiconference on Int. Workshop on MASCOTS-93*, pages 69–78, 1993. La Jolla, California.

[50] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'92)*, pages 1–12, Jun. 1992.

[51] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of VII Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996.

[52] Z. Michalewicz. *Genetic algorithms+Data structures=Evolution Programs*. Springer-Verlag, 1994.

[53] MIPS. *RISCompiler Languages Programmer's Guide*. MIPS, 1988.

[54] A. K. Porterfield. *Software Methods for improvement of cache performance on supercomputer applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[55] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, 1992.

[56] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 121–134, 1994.

[57] B. Rau. Pseudo-randomly interleaved memories. In *Proceedings of International Symposium on Computer Architecture (ISCA'91)*, pages 74–83, 1991.

[58] G. Rivera and C-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.

[59] G. Rivera and C-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of ACM Internacional Conference on Supercomputing (ICS'98)*, 1998.

[60] F.J Sánchez and A. González. Fast, flexible and accurate data locality analysis. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[61] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.

[62] R. Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer designs.* PhD thesis, University of Michigan, 1993.

[63] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94)*, pages 261–271, May 1994.

[64] O. Temam, E.D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing (SC'93)*, pages 410–419, 1993.

[65] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Proceedings of 30th Symposium on Microarchitecture (MICRO-30)*, 1997.

[66] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(3):128–170, Sept. 1997.

[67] E. van der Deijl, G. Kanbier, O. Temam, and E.D. Granston. A cache visualization tool. *IEEE Computer*, 30(7):71–78, July 1997.

[68] X. Vera, J. Llosa, and A. González. Near-optimal padding for removing conflict misses. In *15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02)*, July 2002.

[69] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *Proceedings of European Conference on Parallel Computing (Europar'00)*, 2000.

[70] X. Vera and J. Xue. Analysing cache behaviour for programs with IF statements. Technical Report UNSW-CSE-TR0107, University of New South Wales, May 2001.

[71] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA 8)*, Cambridge, Feb. 2002.

[72] D. A. B. Weikle, K. Skadron, S. A. McKee, and W. A. Wulf. Cache as filters: a unifying model for memory hierarchy analysis. Technical Report CS-2000-16, University of Virginia, Jun. 2000.

[73] R. T. White, F. Müeller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Pro-*

ceedings of Third IEEE Real-Time Technology and Applications Symposium (RTAS'97), pages 192–202, 1997.

[74] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, Oregon State University, 1993.

[75] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, May 1996.

[76] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, Mar. 1992.

[77] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Jun. 1991.

[78] M. Wolfe. Advanced loop interchanging. In *Proceedings of International Conference on Parallel Processing (ICPP'96)*, 1996.

[79] M. E. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[80] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[81] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[82] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.